PALMER: Perception-Action Loop with Memory for Long-Horizon Planning

Onur Beker  Mohammad Mohammadi  Amir Zamir
Swiss Federal Institute of Technology (EPFL)

Abstract
To achieve autonomy in a priori unknown real-world scenarios, agents should be able to: i) act from high-dimensional sensory observations (e.g., images), ii) learn from past experience to adapt and improve, and iii) be capable of long horizon planning. Classical planning algorithms (e.g. PRM, RRT) are proficient at handling long-horizon planning. Deep learning based methods in turn can provide the necessary representations to address the others, by modeling statistical contingencies between observations. In this direction, we introduce a general-purpose planning algorithm called PALMER that combines classical sampling-based planning algorithms with learning-based perceptual representations. For training these perceptual representations, we combine Q-learning with contrastive representation learning to create a latent space where the distance between the embeddings of two states captures how easily an optimal policy can traverse between them. For planning with these perceptual representations, we re-purpose classical sampling-based planning algorithms to retrieve previously observed trajectory segments from a replay buffer and restitch them into approximately optimal paths that connect any given pair of start and goal states. This creates a tight feedback loop between representation learning, memory, reinforcement learning, and sampling-based planning. The end result is an experiential framework for long-horizon planning that is significantly more robust and sample efficient compared to existing methods.

1 Introduction
Animals and humans operate on high-dimensional stimuli (e.g., vision) to achieve diverse and ever-changing goals necessary for their survival [1, 2, 3, 4, 5]. Learning through trial-and-error plays a fundamental role in this [6, 7, 8, 9, 10, 5]. Even in simplest environments, a brute-force approach to trial-and-error by trying every possible action for achieving every possible goal is intractable. The complexity of this search motivates memory-based mechanisms for compositional thinking. Examples of such mechanisms include: i) remembering relevant segments of past experience, ii) recomposing them in new counterfactual ways to form plans, and iii) executing such plans as part of a targeted search strategy. Such mechanisms for recycling past successful behavior can significantly accelerate trial-and-error compared to uniformly sampling all possible actions. This is because the same behavior (i.e., sequence of actions) can remain valid for different goals and in different contexts, due to the inherent compositional structure of real-world goals as well as the commonality of the physical laws that govern real-world environments.

What principles can allow for memory mechanisms to remember and recompose bits of experience? The concept of dynamic programming (DP) is directly related to this discussion, as it greatly reduces the computational cost of trial-and-error by employing the principle of optimality [11]. This principle can be colloquially stated as treating new and complex problems as a recomposition of old and simpler sub-problems that were already solved before. Recent work [12, 13, 14] employs this perspective to
Figure 1: Top: Given a start-goal image pair, PALMER plans a path between them by concatenating the endpoints of past trajectory segments retrieved from a provided replay buffer. This is enabled by a state embedding function $f_{\phi}$ that can identify close-by states, and results in robust long-horizon planning. Bottom: To achieve this: i) it uses offline Q-learning to obtain local reachability estimates between states, ii) uses these Q-values for representation learning to train $f_{\phi}$, iii) uses $f_{\phi}$ to plan over the replay buffer, iv) executes these plans, v) evaluates the resulting trajectories and inserts them back into the replay buffer to improve its contents.

Contribution: We describe a long-horizon planning method that directly operates on high dimensional sensory input observable by an agent on its own (e.g., images from an onboard camera). Our method combines classical sampling-based planning algorithms with learning-based perceptual representations, to retrieve and recompose previously observed sequences of state transitions in a replay buffer. This is enabled by a two-step process. First, we learn a latent space where the distance between two states captures how many timesteps it takes for an optimal policy to go from one to the other. To achieve this, we use goal-conditioned Q-values learned through offline hindsight relabelling [15] for contrastive representation learning. Second, we threshold this learned latent distance metric to define a neighborhood criterion between states. We then define sampling-based planning algorithms that search over the replay buffer [12] to retrieve and stitch together trajectory segments (i.e., past sequences of observed transitions) whose endpoints are neighboring states. This trajectory stitching approach allows for creating planning graphs to connect any pair of start and goal states that were observed before (as depicted in Fig.1). Our approach operates on offline unlabeled data, and can therefore be combined with any exploration method to populate the replay buffer. Our experiments implement an image-based navigation policy in simulation, using an offline replay buffer populated with uniform random-walk exploration data.

2 Perception-Action Loop with Memory Retrieval

Nomenclature: An environment is represented as a tuple $\langle S, A, p_{env} \rangle$, where $S$ and $A$ are the state and action spaces, and $p_{env}(s'|s, a)$ is the Markovian transition dynamics. A trajectory $\tau \in T$ is any sequence of states and actions. $\tau_0$, $\tau_{-1}$, $\tau_i$ denote the first, last, and $i$'th states in $\tau$ respectively. The length of a trajectory in terms of timesteps is denoted as $len(\tau)$, and concatenation of two

---

1 Most sub-sections have a corresponding section in the supplementary for further elaboration.
We propose a model (depicted in Fig. 2) that includes the following standard components from any perceptual encoder $f$.

A key component of our framework is a perceptual encoder $f$ (i.e., how many timesteps it takes for the optimal policy to go from one state to another). To discuss isolation, we freeze its parameters and use it to define a contrastive loss function [21] with a reward function discussed in [18, 19]; modelling the transition distribution induced by $p$.

One possible way for the upcoming sections 2.3 and 2.4. This section discusses in Sec.2.1 can be used to implement the nearest neighbor retrieval and trajectory stitching mechanisms.

2.1 Perceptual Representations that Capture Local Reachability

A key component of our framework is a perceptual encoder $f_\phi(s) : \mathcal{S} \to \mathbb{R}^d$ that maps states into a representation space where L2 distance $d_\phi(s_t, s_g) := \|f_\phi(s_t) - f_\phi(s_g)\|$ captures local reachability (i.e., how many timesteps it takes for the optimal policy to go from one state to another). To discuss this more rigorously, we follow the work of [16] and define a goal-conditioned reward function $r(s_t, a, s_{t+1}, s_g) = -1_{s_{t+1} \neq s_g}$, that returns $-1$ for all steps before reaching a goal. This means goal-conditioned Q-values [16] for the optimal policy correspond to negative shortest-path distances (i.e., $\max_a Q(s_t, a, s_g) = V(s_t, s_g) = -\text{len}(\tau_{sp})$). We can then define a symmetric distance metric between states as $d_Q(s_c, s_g) := \text{max}(V(s_c, s_g), -V(s_g, s_c))$. This in turn corresponds to the two-way consistency criterion proposed in [13]. What we want from $f_\phi(s)$ is for $d_\phi(s_c, s_g)$ and $d_Q(s_c, s_g)$ to roughly correlate.

2.2 Representation Learning via Reinforcement Learning

Any perceptual encoder $f_\phi$ whose latent representations satisfy the local reachability property defined in Sec 2.1 can be used to implement the nearest neighbor retrieval and trajectory stitching mechanisms for the upcoming sections 2.3 and 2.4. This section discusses one possible way to obtain such a perceptual encoder, by using goal-conditioned Q-values for contrastive representation learning.

We propose a model (depicted in Fig 2) that includes the following standard components from the literature: i) $z = f_\phi(s)$, projecting a state into a latent representation; ii) $p_{\text{fwd}}(z'_{t+1} \mid z_t, a_t)$, modelling the transition distribution induced by $p_{\text{env}}(s_t' \mid s_t, a_t)$ over the latent space $z = f_\phi(s)$, as discussed in [13, 19]; iii) $\pi_{\text{inv}}(a'_t \mid z_t, s_g)$, defining a distribution of actions to reach a goal state, as discussed in [18, 19, 14]; iv) $p_i(T \mid z_t, s_g)$, modelling the distribution of timesteps necessary to reach a goal state, as discussed in [20]; v) $Q(s_t, a_t, s_g)$, a Q-value function that provides local reachability estimates between pairs of states, as discussed in [12, 16].

Following [12, 16], we train $Q(s_t, a_t, s_g)$ over an offline replay buffer $\mathcal{M}$, using hindsight relabelling [15] with a reward function $r(s_t, a, s_{t+1}, s_g) = -1_{s_{t+1} \neq s_g}$. After training $Q(s_t, a_t, s_g)$ in isolation, we freeze its parameters and use it to define a contrastive loss function [21] $L_Q$ as explained below. We then train the remaining components using the same replay buffer $\mathcal{M}$. We randomly sample a transition $(s_t, a_t, s_{t+1})$ and a time difference $\bar{T}$, and set the goal state as $s_g := s_{t+T}$, as in hindsight relabelling. We then minimize the following losses:

- $L_Q(s_t, s_g) = L_{\text{hinge}}(d_\phi(s_t, s_g) - d_p) \cdot \mathbb{1}_{d_Q(s_t, s_g) \leq c_Q} + L_{\text{hinge}}(d_p - d_\phi(s_t, s_g)) \cdot \mathbb{1}_{d_Q(s_t, s_g) \geq c_Q}$, where $L_{\text{hinge}}$ is the hinge loss [22]. This contrastive loss dictates that perceptual representations should be close together (i.e., $d_\phi(s_t, s_g) \leq d_p$ holds) if and only if two states are close to each other in terms of reachability (i.e., $d_Q(s_t, s_g) \leq c_Q$ holds). $d_p$ and $c_Q$ are hyperparameters.
• $L_T(T', T)$, $L_{inv}(a_i', a_i)$, and $L_{fwd}(a_i', z_{i+1})$ are MSE and cross-entropy losses \[19\] \[20\]. $L_T$ and $L_{inv}$ dictate that perceptual representations should capture enough information to know when and how an agent can reach from one state to another, while $L_{fwd}$ dictates that they should capture only a minimal-sufficient statistic for doing so (\[19\] presents a more elaborate discussion).

2.3 Perceptual Experience Retrieval (PER)

Given a perceptual encoder $f_{\phi}$ that captures local reachability, we go over all states $s_i \in M$ in the replay buffer and compute their projections $z_i = f_{\phi}(s_i)$, which are stored alongside the states themselves. We then employ $z_i$ to implement two retrieval mechanisms from the replay buffer: i) retrieving neighboring states, and ii) retrieving neighboring trajectories.

i) Retrieving Neighboring States: Given a query state $s_c$ and radius $d_p$ (i.e., the same one used in the contrastive loss $L_Q$ in Sec \[2.2\]), retrieving neighboring states amounts to computing the set $\mathcal{N}_{d_p}(s_c) = \{s_n \mid d_p(s_c, s_n) \leq d_p\}$, which can be achieved by a straightforward L2 distance computation and thresholding. The number of neighbors $|\mathcal{N}_{d_p}(s_c)|$ of a query state $s_c$ is an approximate measure of how many times the agent has visited around $s_c$, which also makes it a good visitation-count that is applicable to both discrete and continuous state spaces.

ii) Retrieving Neighboring Trajectories: Given a starting state $s_c$ and a goal state $s_g$, we can search the replay buffer for the highest reward trajectory segment $\tau$ that starts from a state $\tau_0$ in $\mathcal{N}_{d_p}(s_c)$ and ends in a state $\tau_{-1}$ in $\mathcal{N}_{d_p}(s_g)$. This corresponds to the following optimization problem:

$$\tau_M(s_c, s_g) := \arg \max_{\tau \in \mathcal{M}} R(\tau) \text{ s.t. } \tau_0 \in \mathcal{N}_{d_p}(s_c), \tau_{-1} \in \mathcal{N}_{d_p}(s_g)$$

To find $\tau_M(s_c, s_g)$, we first select all state pairs $(s_i, s_j) \in \mathcal{N}_{d_p}(s_c) \times \mathcal{N}_{d_p}(s_g)$. We then take all sequences of transitions $\tau_{ij} = \{s_i, a_i, s_{i+1}, ..., s_{j-1}, a_{j-1}, s_j\}$ that start from $s_i$, end at $s_j$, and are below a length threshold in terms of timesteps. We sort them based on $R(\tau_{ij})$, and return the trajectory with the highest reward. We call this trajectory retrieval process ‘Perceptual Experience Retrieval’ (PER). We use PER only to retrieve short trajectory segments between close-by states $(s_c, s_g)$ (i.e., hence the length threshold on $\tau_{ij}$). These are then stitched together into long global trajectories using the planning algorithms defined in the next section.

2.4 Long-Horizon Planning Through Stitching Trajectory Segments

This section discusses how PER can be employed for long-horizon planning. Classical sampling-based planning algorithms such as RRT \[23\] or PRM \[24\] connect points sampled from obstacle-free space with line segments in order to build a planning graph. We instead reimagine them as memory search mechanisms by altering their subroutines so that whenever an edge is created, a trajectory is retrieved from the replay buffer through PER (eq\[1\]) and stored in that edge. Our new definitions for these subroutines directly mirror the original ones given in \[25\]:

1) Sampling: Sampling originally returns a point from obstacle free space. We instead return a state $s_c$ from the replay buffer $M$ using any distribution (e.g., uniform, or based on visitation-counts).

2) Lines and Their Cost: The equivalent of drawing a line segment in our framework is retrieving a trajectory $\tau_M(s_c, s_g)$ and its length and cost are $\text{len}(\tau_M(s_c, s_g))$ and $-\mathcal{R}(\tau_M(s_c, s_g))$ respectively.

3) Nearest State and Neighborhood Queries: Given a query point $s_i$, these subroutines return the closest point or a neighborhood of points within a distance, among a set of vertices $V = \{s_j\}$. We preserve these definitions, and only replace the metric from euclidean distance to $\text{len}(\tau_M(s_c, s_g))$.

$$\text{Nearest}(V, s_g) := \arg \min_{s_i \in V} \text{len}(\tau_M(s_c, s_g))$$

$$\text{Near}(V, s_g, r) := \{s_c \in V \mid \text{len}(\tau_M(s_c, s_g)) \leq r\}$$

4) Collision Tests: Collision tests originally prevent the sampling and line drawing subroutines from intersecting obstacles. Since we are planning in retrospect, any such undesirable event can be handled during PER by adjusting the reward function (i.e., if $\tau$ has such an event, this reflects on $\mathcal{R}(\tau)$).

Using these subroutines directly in-place of their originals, we reimplement experiential equivalents of PRM, RRT, and RRT*, which we call R-PRM, R-RRT, R-RRT*. We denote the resulting planned trajectory as $\tau_M^*(s_c, s_g)$. Algorithms \[1\] \[2\] describe R-PRM as an example, and the supplementary contains descriptions for R-RRT, R-RRT*. 

4
Our approach is closely related to goal-reaching methods that combine learning-based distance-regression with graph search, particularly Semi-parametric Topological Memory (SPTM) [14] and Search on the Replay Buffer (SoRB) [12], which we compare to in our experiments. The key difference of our approach is that when setting the edges of the planning graph, it retrieves transitions that actually happened rather than relying on learned distance regression. This brings two main benefits. First is robustness. Local reachability estimates are susceptible to overestimation when evaluated between pairs of states that are far apart or unreachable. This...
Figure 3: A comparison between perceptual distances $d_\phi$ and other suitable metrics from Sec. 2.2. While all of these metrics are reasonably monotonic with physical reachability (i.e., goal distance), only perceptual distances $d_\phi$ do not saturate when evaluated locally (i.e., for close by goals). In addition, the ratio between the variance of $d_\phi$ and the slope of its mean is much smaller compared to other sensible metrics (i.e., $d_\phi$ has a high signal-to-noise ratio). This means that perceptual distances can implement a more accurate nearest-neighbor criterion for perceptual experience retrieval and trajectory stitching, compared to the other metrics.

is because such states rarely occur together and are therefore out of distribution for the distance regression model. This creates ‘hallucinated’ shortcuts in the planning graph that corrupt shortest path queries [12, 13]. To address this, [14] employs temporally consistent localization and adaptive waypoint selection, while [12] employs distributional Q-learning and an ensemble of Q-functions. In our approach, eq.1 naturally addresses this problem, since it requires an actual short trajectory in the dataset approximately connecting two states before marking them as close. The second benefit of our approach is that it can optimize general reward functions. This is because it decouples the reachability metric $\text{len}(\tau)$ (used in nearest neighbor queries and as a threshold to create edges) from the downstream task reward $R(\tau)$ (used to set edge distances), unlike previous work.

Image-Based Navigation: [26, 27, 28] present learning-based navigation systems that incrementally build roadmaps through online operation. Our approach has two main differences: i) it builds a roadmap entirely using raw offline data, therefore allowing applications like multi-robot learning without additional loop-closure mechanisms to fuse graphs from multiple agents, ii) our approach can optimize general reward functions, therefore it is not limited to navigation.

Robot Motion Planning: A common approach to motion planning is to first run a sampling-based planning algorithm [29, 25], and then refine the result through trajectory optimization [30, 31, 32] to satisfy constraints [33, 34, 35]. An important bottleneck is that sampling-based planning algorithms require a precomputed map of the environment, and our approach extends such algorithms in a way that relaxes this requirement by replacing a precomputed map with raw exploration experience.

SLAM and Geometric Maps: SLAM based methods [36] can autonomously construct high-fidelity geometric maps [37, 38], therefore alleviating the bottleneck of precomputing environment maps. The downside of such approaches is that they can abstract away useful physical and semantic affordances. For example, a purely geometric map cannot plan a path through a traversable field of tall-grass, while our approach can learn such affordances as long as they are represented in past experiences.

4 Experiments  

Setup: Our experiments are performed in ViZDoom [39], Habitat [40], and the Maze2D benchmark [41]. The VizDoom environment consists of a clover shaped maze. States solely consist of four images $I_{\text{North/East/South/West}}$ that form a panorama (i.e., $4 \times 3 \times 160 \times 120$ dimensions), and actions move the agent North/South/East/West by a fixed distance $\Delta$. The maze contains many long-thin column-like obstructions (shown as dots in visualizations). Habitat experiments contain demonstrations on two large-scale scans of real-world apartments: i) Roxboro, with a total area of 62 m$^2$, and ii) Annawan, which has a total-area of 75m$^2$. States consist of a single 150 FOV image (i.e., $3 \times 256 \times 256$ dimensions). There are 3 actions: \{turn_left 30 deg, turn_right 30 deg, move_forward $\Delta$\}. Maze2D is a continuous control task, where states consist of the 2D position and velocity of a point mass, and actions correspond to 2D accelerations. In all environments, an offline training dataset is collected by a uniform random walk exploring the environment. For VizDoom and Habitat, this offline training dataset consists of only 300k and 150k timesteps respectively, while for Maze2D there are 1e6 timesteps. Supplementary material contains further details.

\footnote{All experiments have a corresponding section in the supplementary providing further implementation details.}
Figure 4: Comparisons of our local policy $\pi_M$ and global policy $\pi_M^*$ with SPTM and SoRB. $\pi_M$ performs well because it avoids getting stuck (as such events are filtered by eq.1), while $\pi_M^*$ performs well because it builds robust roadmaps without hallucinated shortcuts; therefore avoiding the main failure modes of the baselines.

Figure 5: At the core of PALMER is a process called perceptual experience retrieval (PER). Given a query pair of current-goal states, PER searches the replay buffer to retrieve the highest scoring trajectory $\tau_M(s_t,s_g)$ whose first and last states are close to the query pair according to the perceptual distance $d_\phi$. Left, Middle: Visualizations of $\tau_M(s_t,s_g)$ retrieved using PER and nearest neighbor states $N_{d_\phi}(s_t)$ retrieved using $d_\phi$. Right: Setting edges of a roadmap using $\text{len}(\tau_M(s_t,s_g))$, compared with distance estimates used in SORB and DDL [20]. We found that distance estimates from baselines are prone to setting false edges that cross map boundaries.

4.1 Experiments in Vizdoom

Validating Perceptual Representations: Fig.3 shows that $d_\phi(s_t,s_g)$ obtained from our model captures a suitable notion of local reachability. Fig.5 in turn shows that retrieving nearest neighbor states $N_{d_\phi}(s_t)$ from $M$ using $d_\phi$ (i.e., NN retrieval) returns physically close states.

Validating Perceptual Experience Retrieval (PER): Fig.5 shows visualizations of trajectories retrieved with PER. We implement a retrieval policy $\pi_M$ that computes $\tau_M(s_t,s_g)$ through eq.1 at each timestep $t$ and executes $\text{argmax}_a Q(s_t,a;\tau_M(s_t,s_g),s,1)$, therefore forming a model predictive control (MPC) loop. We evaluate $\pi_M$ in an image-based navigation task where start/goal images are sampled randomly to have an euclidean distance $n \times \Delta$ in between, and a trial is considered successful if the agent can get within $\Delta$ proximity of the goal position within $4 \times n$ time-steps. We use the local policies from SORB [12] and SPTM [14] as baselines. Fig.4 shows the results. The main mode of failure for both SPTM and SORB local policies is that they get stuck in column-like structures. $\pi_M$ avoids this, since eq.1 retrieves collision free $\tau_M(s_t,s_g)$.

Robust Distances: PER also helps avoid hallucinations in local distance regression. Fig.5 illustrates this point by setting edges between sampled states by thresholding $\text{len}(\tau_M(s_t,s_g))$, where methods of [12,20] are used as baselines. It can be seen that edges set by $\text{len}(\tau_M(s_t,s_g))$ are more robust.

Proposed Planning Algorithms: Fig.6 shows visualizations of planning graphs and $\tau_M^*(s_t,s_g)$ produced by R-PRM, R-RRT, and R-RRT*. It can be seen that R-PRM doesn’t contain any hallucinated edges, while R-RRT and R-RRT* maintain the visual characteristics of their classical counterparts (i.e., R-RRT has jagged branches with uniform coverage, while R-RRT* has straight branches shooting out from the root). We implement an MPC policy $\pi_M^*$ that replans at each timestep $t$ using Algorithm 2 to return $\tau_M^*(s_t,s_g)$, and executes $\text{argmax}_a Q(s_t,a;\tau_M^*(s_t,s_g),s,1)$. We again...
Figure 6: We repurpose conventional sampling-based planning algorithms as memory search mechanisms, by altering their graph building subroutines so that whenever an edge is created a trajectory $\tau_{M}(s,t,s_{g})$ is retrieved through PER and stored in that edge. We visualize the resulting planning graphs produced by our proposed algorithms R-PRM, R-RRT, R-RRT*.

Figure 7: Memory Refinement: In PALMER, a policy has three groups of parameters: $Q(s,t,s_{g})$, $f_{\phi}$, and the contents of $M$. Iteratively forming plans through PER and executing them creates a feedback loop between these components, where: i) actions inform perception during the training of $f_{\phi}$, ii) perception facilitates actions through the formation plans, and iii) memory serves as the medium for this reciprocal interaction. As a result, trajectories produced by explicit planning are gradually internalized as implicit behavior encoded in the model parameters. This leads to: Q-values propagating further into distant goals (Left), memory contents getting closer to optimal (Middle), and performances of local policies showing significant improvement (Right).

use SORB \cite{12} and SPTM \cite{14} as baselines. Fig.4 shows the results. In addition to the local policy getting stuck, a new mode of failure for both baselines is that false distance estimates throw-off graph search by setting hallucinated shortcuts. A new baseline is $\pi_{mpc}$, which extends the SPTM local policy by using $p_{fwd}$ and $p_{t}$ from Sec.2.2 to implement an MPC loop with n-step look-ahead. $\pi_{mpc}$ avoids getting stuck in columns thanks to n-step lookahead, but still isn’t sufficient for global navigation as the accuracy of simulated rollouts from $p_{fwd}$ decreases with the number of timesteps.

Refining Memory Contents: We refine the contents of $M$ by iteratively generating and executing $\tau_{M}^{*}(s_{c},s_{g})$. We then retrain all model components only on the resulting new data that is equal in size to the initial unrefined $M$. Fig.7 shows the results. When $\pi_{M}$, and $argmax_{a}Q(s_{c},a,s_{g})$ are used as policies, their success ratio increases significantly if they are trained on the optimized $M$. Q-value estimates trained on the optimized $M$ also propagate better to goals further away. The scaling of $len(\tau_{M}(s_{c},s_{g}))$ with goal-distance changes from an exponential trend to an approximately linear one, due to the inclusion of transitions from successfully executed $\tau_{M}^{*}(s_{c},s_{g})$. These results highlight that refining memory contents improves the quality of future plans.

4.2 Experiments in Habitat

As shown in Fig.8, we find that our method allows image-based navigation in this new domain with significantly different visuals and layouts (i.e., real-world apartments), action space (i.e., turn-left, turn-right, go-forward), and state space (i.e., single $256 \times 256$ RGB images with 150 FOV). Perhaps more surprisingly, we find that training $f_{\phi}$ only on exploration data from a single apartment generalizes substantially well to any unseen apartment, which directly allows perceptual experience retrieval and trajectory stitching when provided with a corresponding replay buffer. For a quantitative evaluation, we randomly pick two apartments, named Roxbox and Annawan. In both apartments, we collect an exploration dataset using a uniform random walk sequence of only 150k timesteps. We

---

\textsuperscript{3}For a comparison without confounders, we train SoRB with DDQN \cite{42} rather than distributional Q-learning \cite{43}, and we do not employ temporally consistent localization for SPTM, as such fixes are equally applicable to our method and orthogonal to the discussion. The supplementary provides further elaboration.
Figure 8: We evaluate our R-PRM based policy $\pi^*_M$ in the Habitat simulator for image-based navigation. 

**Top Left:** Success ratios in training and test apartments. **Top Right:** Number of timesteps until reaching the goal. 

"Habitat seen" refers to the training apartment Roxbox, while "habitat unseen" refers to the test apartment Annawan. **Bottom:** We found that training the perception model $f_\phi$ on a **single** apartment generalizes sufficiently well to allow perceptual experience retrieval and trajectory stitching in any unseen apartment.

To test our method on a continuous control task, we perform additional experiments on the Maze2D benchmark. As shown in Table 1, we find that the same $\pi^*_M$ policy from sections 4.1 and 4.2 achieves strong performance, and can solve mazes of all three complexities.

Table 1: Total rewards on the Maze2D benchmark, which is a continuous control task that requires long-horizon planning. Our R-PRM based $\pi^*_M$ policy achieves comparatively strong performance.

<table>
<thead>
<tr>
<th>Maze2D</th>
<th>SAC</th>
<th>SAC-off</th>
<th>BEAR</th>
<th>AWR</th>
<th>BCQ</th>
<th>CQL</th>
<th>IQL</th>
<th>Diffuser</th>
<th>PALMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>maze2d-umaze</td>
<td>110.4</td>
<td>145.6</td>
<td>28.6</td>
<td>25.2</td>
<td>41.5</td>
<td>31.7</td>
<td>89.6</td>
<td>182.1</td>
<td>131.76</td>
</tr>
<tr>
<td>maze2d-medium</td>
<td>69.5</td>
<td>82.0</td>
<td>89.8</td>
<td>33.2</td>
<td>35.0</td>
<td>26.4</td>
<td>105.2</td>
<td>332.9</td>
<td>416.28</td>
</tr>
<tr>
<td>maze2d-large</td>
<td>14.1</td>
<td>1.3</td>
<td>19.0</td>
<td>70.1</td>
<td>23.2</td>
<td>40</td>
<td>159.9</td>
<td>328.1</td>
<td>361</td>
</tr>
</tbody>
</table>

5 Discussion and Future Directions

*Is PALMER less expressive than standard deep Q-learning? Two important premises of deep Q-learning [44, 34] are: i) minimizing Bellman error through temporal-difference (TD) updates can restitch observed transitions in new optimal ways [41, 45], ii) a neural network can learn to extrapolate
Q-values to unobserved but close-by states in high-dimensional spaces (e.g., images) [46]. Both arguments are equally valid for our approach, since it can: i) restitch transitions at arbitrary resolutions (i.e., anywhere from one-step transitions to multi-step trajectories) by virtue of sampling-based planning, ii) group together close-by states through $d_\phi$. Therefore, PALMER is an RL algorithm that: i) optimizes Bellman error through sampling-based optimal planning rather than gradient-based TD-updates [46], ii) performs extrapolation between states using a perceptual-backbone $f_\phi$ rather than a deep Q-network, and iii) replaces the greedy-policy $\arg\max_{a} Q(s_t, a, s_g)$ and value estimate $\max_{a} Q(s_t, a, s_g)$ with $\arg\max_{a} Q(s_t, a, \tau_{M^*(s_t, s_g), s, t})$ and $R(\tau_{M^*(s_t, s_g)})$ respectively. The key benefits of these alterations come into play when $s_t$ and $s_g$ are far apart, and these benefits are: i) the PER mechanism in eq.1 that prevents hallucinations in $Q(s_t, a, s_g)$, ii) global propagation of value estimates by virtue of employing sampling-based planning methods, which are known to be particularly proficient at searching high-dimensional state spaces across long-horizons [35][29].

Combining PALMER with standard deep Q-learning: Our approach can also be flexibly combined with any traditional Q-learning method [46][47][48], by using our proposed planning algorithms (Sec.2.4) as experience replay methods [49]. This alternative approach stitches together $\tau_{M^*(s_t, s_g)}$ during training, and perform backwards TD-updates over this trajectory starting from $s_g = \tau_{M^*(s_t, s_g), s, t}$ and ending at $s_c = \tau_{M^*(s_t, s_g), 0}$. As suggested by Fig.7, this can allow value estimates $Q(s_t, a, s_g)$ to propagate more globally. Our proof-of-concept experiments identify this as a promising direction, and we leave a further extensive evaluation to future work.

Connections to contingency learning: Contingency learning refers to the acquisition of knowledge of statistical correlations between percepts [50][3][51]. Following this definition, PALMER can be interpreted as a contingency learning framework, as the latent distance metric $d_\phi$ captures statistically how likely two states are to be observed in close temporal proximity. The knowledge of these statistical contingencies between states is then used for long-horizon decision making through the proposed perceptual experience retrieval and planning mechanisms.

6 Conclusion and Limitations

We presented PALMER, a long-horizon planning method that combines learning-based perceptual representations with classical sampling-based planning algorithms. Given a goal state $s_g$ and reward function $R$, our method searches the contents of an offline replay-buffer $M$ to stitch together a sequence of transitions $\tau_{M^*(s_t, s_g)} = \{s_1, a_1, s_2, \ldots\}$ that reaches $s_g$ while maximizing $R$. This results in an experiential framework for long-horizon planning that is significantly more robust and sample efficient compared to baselines.

Our experiments show that PALMER can successfully solve long-horizon planning tasks from continuous high-dimensional inputs. In particular, we have shown that given an offline dataset of only 150k transitions (i.e., compared to sample complexities around the orders of magnitude $10^6$-10^7 common in RL) obtained from an entirely uniform random-walk (i.e., which is significantly less structured compared to on-policy rollouts), it allows image-based navigation between any two points in large-scale scans of real-world apartments.

We believe that our memory-based planning perspective highlights a number of interesting questions for future research. First, which transitions should be kept in the replay buffer $M$, and which ones should be discarded? $M$ cannot be infinitely expanded after deployment, and it is critical to distill away redundancies between stored experiences. Second, when the environment undergoes a change, which transitions in the replay buffer remain valid and can still be used for planning, and which ones become invalid? A mechanism that can answer this question can allow quick and sample-efficient adaptation to environmental changes. Third, how can we extend $f_\phi$ to allow more abstract associations and functional equivariances between states? This can improve generalization by defining a more flexible notion of experience retrieval that can recycle past behavior in new contexts and for new tasks. We leave these questions to future work.

References


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]
Supplementary Material

This document presents additional implementation details, visualizations, and conceptual discussions that were excluded or briefed due to space limitations in the main paper. The organization mirrors the sections from the main paper (with the exact same order and numbering), and the exposition generally maintains a question-answer format. It aims to provide significantly more details, at a degree of clarity sufficient for re-implementation. As such, we recommend referring to this document whenever any section in the main paper can benefit from further elaboration. The supplementary material consists of:

• vizdoom_nav — navigation videos in VizDoom.
• habitat_roxbox_nav — navigation videos in the training apartment Roxbox in Habitat.
• habitat_annawan_nav — navigation videos in the test apartment Annawan in Habitat.
• maze2d_experiments — goal reaching videos in Maze2D.
• planning_vis — videos visualizing the execution of R-RRT and R-RRT*.
• code — source code for PALMER.

Table of Contents

1. Introduction
   • Why learn a latent distance metric for nearest neighbor retrieval?
   • What does stitching transitions together mean?

2. Perception-Action Loop with Memory Reorganization
   2.1. Perceptual Representations that Capture Local Reachability
       • What does local reachability mean?
   2.2. Representation Learning via Reinforcement Learning
       • How are the model components trained, and what are their exact inputs and outputs?
       • What do the terms in the contrastive loss-function $L_Q$ mean?
   2.3. Perceptual Experience Retrieval (PER)
       • How do we solve the optimization problem in equation 1 from the main paper?
       • What do the retrieved trajectories $\tau_M(x_c, x_g)$ look like?
   2.4. Long-Horizon Planning Through Stitching Trajectory Segments
       • How does R-PRM work in detail?
       • How does R-RRT work in detail?
       • How does R-RRT* work in detail?
       • What does optimizing the Bellman error on the roadmap mean?
       • What does restitching transitions at arbitrary resolution mean?
   2.5. Refining Memory Contents via Forming and Executing Plans
       • What does optimizing memory contents mean?
       • How does the perception-action loop in PALMER work in detail?

3. Related Work
   • What is the main reason why memory-based reasoning over actually observed transitions is necessary? Why are methods like SPTM or SoRB that solely rely on learning-based distance estimates are inherently prone to false predictions?
4. Experiments

Setup

- What are the details for the experimental setup in VizDoom?

Validating Perceptual Experience Retrieval (PER)

- What is the exact evaluation process that produced Fig.4 in the main paper?

Robust Distances

- What is the exact evaluation process for the right panel of Fig.5 in the main paper?

Proposed Planning Algorithms

- Why does the policy $\pi_{A^*}$ use R-PRM for planning?
- What are the details for the $\pi_{mpc}$ baseline?

Experiments in Habitat

- What are the details for the experimental setup in Habitat?
- Why does the agent occasionally take random-looking actions in the habitat navigation trials?

5. Discussion and Future Directions

- How is PALMER related to the "Options Framework (Sutton et al.)" and "Skill-Chaining (Konidaris et al.)"?
- How is PALMER related to "LQR-Trees (Tedrake et al.)"?

1 Introduction

Why learn a latent distance metric for nearest neighbor retrieval: In a low-dimensional state space such as 3D positions, L2 distance (i.e., euclidean distance) directly correlates with local physical-reachability (i.e., we emphasize local, because euclidian distances still do not match geodesic distances globally). Therefore in such state-spaces, grouping together two nearby states and treating them as the same single state for downstream global planning should still result in a feasible planned trajectory. By feasible, we mean that the gaps and approximations introduced by state grouping are functionally inconsequential and can be handled reasonably well by a local policy tracking the global planned trajectory. This property doesn’t hold in high-dimensional state spaces such as images, since the L2 distance doesn’t correlate with physical reachability. The main purpose of $f_\phi$ is to project such high-dimensional spaces into a low-dimensional representation space where this property holds, so that nearby states can be fused together to make sampling-based planning computationally tractable.

![Diagram](image)

Figure 9: Visualization of stitching together trajectories. If an agent has previous experience of separately going through segments $(A, B)$ and $(C, D)$, it should be able to go from $s_c$ to $s_g$ through the segments $(A, D)$.

What does stitching transitions together mean: As shown in Fig.9 if there are two separate sequences of transitions in an offline memory buffer that traverse segments $(A, B)$ and $(C, D)$, an agent should be capable of going from $s_c$ to $s_g$ through the segments $(A, D)$ even if such a direct path of transitions is not present in the buffer.
was never actually observed. Traditional deep Q-learning methods achieve this by combining and propagating value estimates through TD-updates (i.e., \( \arg\max_a Q(s_t, a, s_{g}) \)) points to segment \( D \) after TD updates over the path \((C, D)\), therefore the argmax policy would follow the path \((A, D)\) when going from \( s_t \) to \( s_{g} \). \cite{41} provides a further discussion). In our approach, this is achieved by setting edge distances for \((A, B, C, D)\) in a planning graph through perceptual experience retrieval, and then performing a shortest path computation to retrieve the path \((A, D)\).

## 2 Perception-Action Loop with Memory Reorganization

### 2.1 Perceptual Representations that Capture Local Reachability

**What does local reachability mean?** At a high-level (and for the special case of image-based navigation) what the term “local reachability” intends to convey is that if two images \( I_1 \) and \( I_2 \) are from physically close positions, \( d_p = |f_p(I_1) - f_p(I_2)| \) should be small. This in turn provides a metric for grouping together states that is better than the L2 distance in image space, which has no correlation with physical reachability. Such a metric is necessary to make search and sampling-based planning planning over the memory buffer computationally tractable. This learned metric \( d_p \) serves the exact same purpose as the hand-crafted image compression criterion employed in \cite{52} to initialize cells from states.

### 2.2 Representation Learning via Reinforcement Learning

**How are the model components trained, and what are their exact inputs and outputs:** More detailed descriptions for all model components are given below (Fig.10 presents a visualization):

- The perceptual backbone \( f_p(s) \) serves the exact purpose as the hand-crafted image compression criterion employed in \cite{52} to initialize cells from states.
- The architecture and training of \( Q(s_t, a_t, s_{g}) \) is completely decoupled from the other components. It consists of cascaded convolutional and fully-connected layers with batch normalization and ReLU activations between each layer. It takes as input the concatenated images for current and goal states (i.e., shape \( B \times C \times H \times W \)), and outputs a vector of Q-values for each action (i.e., shape \( B \times \text{num-actions} \)). It is trained through offline DDQN \cite{42} with hindsight goal-relabelling \cite{15}. We first sample \( t \sim \text{Uniform}(0, \text{dataset-size}) \) and \( T \sim \text{Geom}(p) \), and then retrieve from the replay buffer a transition and a goal state as \((s_t, a_t, s_{t+1}, s_{g} := s_{t+T})\). We then minimize the TD error \( |Q_p(s_t, a_t, s_{g}) - (1 - \gamma) \mathbb{I}_{s_{t+1} = s_{g}} + \gamma \mathbb{I}_{s_{t+1} \neq s_{g}} \max_a Q_p(s_{t+1}, a, s_{g})| \)^2, as in \cite{53}.
- The perceptual backbone \( f_p(s) \) uses a standard Resnet-18 architecture. It takes as input the images for a given state (i.e., \( B \times C \times H \times W \)), and outputs a low-dimensional representation \( z = f_p(s) \) (i.e., shape \( B \times D \)). All other components take as input these low-dimensional representations, rather than operating over images.
- \( p_{\text{fwd}}(z'_{t+1} \mid z_t, a_t), \pi_{\text{inv}}(a'_t \mid z_t, z_{g}), \) and \( p_t(T' \mid z_t, z_{g}) \) all consist of fully-connected layers with ReLU activations. To train them, we first sample \( t \sim \text{Uniform}(0, \text{dataset-size}) \). We then sample \( T \) according to \( T \sim \text{Uniform}(0, T_{\text{max}}) \) or \( T \sim \text{Uniform}(T_{\text{max}}, \text{dataset-size} - t) \), half the time from the former distribution, half the time from the latter. We retrieve from the replay buffer a transition and a goal state as \((s_t, a_t, s_{t+1}, s_{g} := s_{t+T})\), and project them into low dimensional representations \((z_t, a_t, z_{t+1}, z_{g})\) using \( f_p \). These are concatenated and passed to models \( p_{\text{fwd}}(z'_{t+1} \mid z_t, a_t), \pi_{\text{inv}}(a'_t \mid z_t, z_{g}), p_t(T' \mid z_t, z_{g}) \) in a way compatible with their

![Figure 10: Visualization of the model architecture, reproduced here for ease of reference.](image)
The loss function $L_{\text{inv}}$ outputs a vector of probabilities over actions (i.e., shape $B \times \text{num\_actions}$), and is trained with the cross-entropy loss $L_{\text{inv}}$ with $a_t$ as the target. $p_t$ also outputs a discrete probability distribution over $[0, T_{\text{max}}]$ that predicts the distribution of time-steps to reach the goal, where the last bin $T_{\text{max}}$ serves as a catch-all for all values above it. It is trained using the cross-entropy loss $L_T$, with $T$ as the target. As mentioned in the main paper, all components $f_\phi$, $p_{\text{fwd}}(z_{t+1} \mid z_t, a_t)$, $\pi_{\text{inv}}(d'_t \mid z_t, z_g)$, $p_t(T' \mid z_t, z_g)$ are trained jointly, with an additional loss function $L_Q$ that regularizes $f_\phi$.

What do the terms in the contrastive loss-function $L_Q$ mean: The loss function $L_Q(s_t, s_g) = l_{\text{hinge}}(d_\phi(s_t, s_g) - d_p) 1_{d_Q(s_t, s_g) \leq c_Q} + l_{\text{hinge}}(d_p - d_\phi(s_t, s_g)) 1_{d_Q(s_t, s_g) \geq c_Q}$ consists of two penalty terms $l_{\text{hinge}}(d_\phi(s_t, s_g) - d_p)$ (i.e., only active when $d_\phi \geq d_p$) and $l_{\text{hinge}}(d_p - d_\phi(s_t, s_g))$ (i.e., only active when $d_\phi \leq d_p$). These penalty terms are gated through two complementary indicator functions $1_{d_Q(s_t, s_g) \leq c_Q}$ and $1_{d_Q(s_t, s_g) \geq c_Q}$. This essentially means that for $L_Q$ to be zero, $d_\phi \leq d_p$ should hold (i.e., perceptual representations are close) if and only if $d_Q(s_t, s_g) \leq c_Q$ holds (i.e., states are physically close). The reason for employing such a conservative switching mechanism with a hinge loss in $L_Q$ (i.e., rather than a continuous penalty term as in $\text{[21][54]}$) is because Q-value estimates are quite inaccurate (especially when $s_t$ and $s_g$ are far apart), and their exact value is generally unreliable (i.e., they can indicate whether two-states are close sufficiently well, but cannot robustly answer how close they are). To pick the hyperparameter $c_Q$, we compute the average Q-value between states in the replay buffer that were observed to be within one-step proximity, and use a fraction of this value to as a conservative estimate. While conceptually the choice for the hyperparameter $d_p$ is arbitrary, we heuristically pick it by examining the average $d_\phi$ distance between subsequent states in the replay buffer, obtained from a preliminary $f_\phi$ backbone trained without $L_Q$.

2.3 Perceptual Experience Retrieval (PER)

How do we solve the optimization problem in equation 1 from the main paper: Our experiments use $-R(t) = \text{lcn}(t)$. We first compute the perceptual representations $z$ for all states in the replay buffer, and stack them into a tensor block of shape $(\text{dataset\_size}, D)$. Given $s_c$ and $s_g$, we search this tensor with vectorized masking operations to retrieve a set of neighboring states $N(s_c, d_p)$ and $N(s_g, d_p)$ (i.e., sets of states within a perceptual distance threshold $d_p$ of $s_c$ and $s_g$), to address cons.4. We sort the resulting pairs of states $(s_i, s_j) \in N(s_c, d_p) \times N(s_g, d_p)$ in terms of $j - i$, and filter these pairs using cons.5. We then pick the first (i.e., closest) pair, and return all the states with indices between $i, j$ from the replay buffer as the resulting trajectory $\tau_M(s_c, s_g)$. The important thing to emphasize about all of these operations is that they can be trivially vectorized, and therefore the optimization problem in eq.3-5 can be solved in less time than a forward pass of $f_\phi$.

![Figure 11: Visualization of retrieved trajectories, with different perceptual distance threshold. Query states $s_c$ and $s_g$ are represented as white squares with a cross in the center, while the start and end points of the retrieved trajectory $\tau_M(s_c, s_g)$ are denoted with a black square and a yellow square with a diagonal dash respectively.](image)

What do the retrieved trajectories $\tau_M(s_c, s_g)$ look like: As the perceptual distance threshold for $d_\phi$ increases, the physical radii spanned by the nearest neighbor sets $N(s_c, d_p)$ and $N(s_g, d_p)$ increase, as shown in Fig[11]. As a result, constraint 4 in the PER equation gets looser, more trajectories satisfy constraints 4 and 5 (because their start and end points are allowed to deviate further from the query pair $s_c, s_g$), and therefore the optimization in equation 3 returns a shorter trajectory $\tau_M(s_c, s_g)$. 

18
Algorithm 3: Classic PRM (Roadmap Construction)
1: \( V \leftarrow \{\text{SampleFree}_i\}_{i=1,\ldots,\text{num_vertices}}; E \leftarrow \emptyset \) \hspace{1cm} \triangleright \text{Initialize vertices and edges}
2: \text{for each } s_i \in V \text{ do}
3: \quad U \leftarrow \text{Near}(V, s_i, r) \setminus \{s_i\}
4: \quad \text{for each } s_j \in U \text{ do}
5: \quad \quad \text{if } \text{CollisionFree}(s_i, s_j) \text{ then } E \leftarrow E \cup \{(s_i, s_j), (s_j, s_i)\}
6: \quad \text{return } G = (V, E)
\hspace{1cm} \triangleright \text{Draw lines as edges}

Algorithm 4: Classic PRM (Shortest-Path Queries Over the Roadmap)
1: \text{Input: } s_c, s_g, G = (V, E), R(\tau), f_\emptyset, M
2: \text{for each } s_i \in V \text{ do}
3: \quad \text{if } \text{CollisionFree}(s_i, s_c) \text{ then}
4: \quad \quad E \leftarrow E \cup \{(s_c, s_i), (s_i, s_c)\}
5: \quad \text{if } \text{CollisionFree}(s_i, s_g) \text{ then}
6: \quad \quad E \leftarrow E \cup \{(s_i, s_g), (s_g, s_i)\}
7: \quad \text{return } \{s_j\} \leftarrow \text{ShortestPath}(s_c, s_g, G)
\hspace{1cm} \triangleright \text{Shortest path through graph search}

Algorithm 5: R-PRM (Roadmap Construction)
1: \text{Input: } f_\emptyset, M
2: \quad V \leftarrow \{\text{SampleFree}_i\}_{i=1,\ldots,\text{num_vertices}}; E \leftarrow \emptyset \hspace{1cm} \triangleright \text{Initialize vertices and edges}
3: \quad \text{for each } s_i \in V \text{ do}
4: \quad \quad U \leftarrow \text{Near}(V, s_i, r) \setminus \{s_i\}
5: \quad \quad \text{for each } s_j \in U \text{ do}
6: \quad \quad \quad \text{if } \text{CollisionFree}(s_i, s_j) \text{ then}
7: \quad \quad \quad \quad E \leftarrow E \cup \{(s_i, s_j), (s_j, s_i)\}
8: \quad \quad \quad \text{return } G = (V, E)
\hspace{1cm} \triangleright \text{Place PER trajectories in edges}

**How does R-PRM work in detail:** Alg.5-6 give step-by-step descriptions for the classical PRM algorithm (adapted from [25]), while Alg.7-8 describe our new definitions for R-PRM. It can be seen that there are two main differences:

- In R-PRM, whenever an edge is created, a trajectory \(M_{c,s_i}\) is retrieved through perceptual experience retrieval and stored in a field \(\tau_{edge}\), while its reward \(−R(M_{c,s_i})\) is stored in a different field \(d_{edge}\).

- In PRM the length and cost of a line segment are the same (i.e., euclidian distance), whereas in R-PRM the length of a trajectory \(\text{len}(M_{c,s_i})\) and its reward \(−R(M_{c,s_i})\) are decoupled. This means that a shortest path query in R-PRM returns a sequence of nodes and edges that optimize the reward function \(R\). An additional step in R-PRM is that at the end of the shortest-path query, all trajectories \(\tau_{edge} = M_{c,s_i}\) stored in the returned edges are concatenated into a single trajectory \(M'_{c,s_i}\) = \(\tau_{stitched}\).

**How does R-RRT work in detail:** Alg.7 gives a step-by-step description for the classical RRT algorithm (adapted from [25]), while Alg.8 describes our new definition for R-RRT. In addition to the two previous differences between PRM and R-PRM, there is one additional difference between RRT and R-RRT. In RRT, there is a steering sub-routine that draws a line segment of length \(r\) starting from \(s_{nearest}\) and extending towards \(s_{rand}\), to create a new vertex \(s_{new}\). In R-RRT, this is replaced by retrieving a trajectory \(M_{s_{nearest},s_{rand}}\) starting from \(s_{nearest}\) and ending at \(s_{rand}\), and its \(r\)'th state is used to create the new vertex \(s_{new}\).

**How does R-RRT* work in detail:** Alg.9 gives a step-by-step description for the classical RRT* algorithm (adapted from [25]), while Alg.10 describes our new definition for R-RRT*. R-RRT* almost exactly maintains the tree rewiring machinery employed in RRT*, the only difference being
Dynamic programming based graph-based retrospective-planning algorithms R-PRM, R-RRT, R-RRT* are sampling based. This means that if two consequent states \( s_i \) and \( s_{i+1} \) are retrieved during their \texttt{SampleFree} \(_i\) routines, then these algorithms will set an edge using \( \tau_{M(s_i,s_{i+1})} \), which is simply the single transition \( \langle s_i, a_i, s_{i+1} \rangle \). Therefore, given enough samples, these algorithms can restitch trajectories down to the level of individual transitions.

### 2.5 Refining Memory Contents via Forming and Executing Plans

What does optimizing memory contents mean: A replay buffer \( \mathcal{M} \) is a collection of trajectories. Optimizing its contents means adding trajectories to \( \mathcal{M} \) that achieve higher total reward.

How does the perception-action loop in PALMER work in detail: Alg. [11] gives a step-by-step description of the overall perception-action loop implemented in PALMER. What PALMER does is essentially bridging any auxiliary exploration method with any auxiliary exploitation method, by reorganizing exploration experience in \( \mathcal{M} \) into \( \tau_{M'(s_c,s_g)} \) that can be used for exploitation. The particular way in which the trajectories \( \tau_{M'(s_c,s_g)} \) can be executed for exploitation has a great deal of flexibility, for example: all actions in \( \tau_{M'(s_c,s_g)} \) can be executed sequentially in an open-loop manner, the first actions of \( \tau_{M'(s_c,s_g)} \) can be executed in a model predictive control (MPC) manner, states in \( \tau_{M'(s_c,s_g)} \) can be tracked by an auxiliary local feedback controller, or the entirety of \( \tau_{M'(s_c,s_g)} \) can be used to initialize a separate trajectory optimization method.
Algorithm 8 R-RRT
1: \( V \leftarrow \{s_{\text{init}}\}; \; E \leftarrow \emptyset \) \quad \triangleright \text{Initialize vertices and edges}
2: \text{for } i = 1, \ldots, n \text{ do}
3: \quad s_{\text{rand}} \leftarrow \text{SampleFree}_i \quad \triangleright \text{Sample random vertex}
4: \quad s_{\text{nearest}} \leftarrow \text{Nearest}(V, s_{\text{rand}}) \quad \triangleright \text{Find the nearest vertex in } V
5: \quad s_{\text{new}} \leftarrow \tau_M(s_{\text{nearest}}, s_{\text{rand}})^r \quad \triangleright \text{Get the } r \text{ \textsuperscript{th} state in } \tau_M(s_{\text{nearest}}, s_{\text{rand}})
6: \text{if } \text{len}(\tau_M(s_{\text{nearest}}, s_{\text{new}})) \leq r \text{ then}
7: \quad V \leftarrow V \cup \{s_{\text{new}}\}
8: \quad E \leftarrow E \cup \{(s_{\text{nearest}}, s_{\text{new}}) : \tau_{\text{edge}} = \tau_M(s_{\text{nearest}}, s_{\text{new}}),
9: \quad \quad d_{\text{edge}} = -R(\tau_M(s_{\text{nearest}}, s_{\text{new}}))\}
10: \text{return } G = (V, E)

Algorithm 9 Classic RRT*
1: \( V \leftarrow \{s_{\text{init}}\}; \; E \leftarrow \emptyset \) \quad \triangleright \text{Initialize vertices and edges}
2: \text{for } i = 1, \ldots, n \text{ do}
3: \quad s_{\text{rand}} \leftarrow \text{SampleFree}_i \quad \triangleright \text{Sample random vertex}
4: \quad s_{\text{nearest}} \leftarrow \text{Nearest}(V, s_{\text{rand}}) \quad \triangleright \text{Find the nearest vertex in } V
5: \quad s_{\text{new}} \leftarrow \text{Steer}(s_{\text{nearest}}, s_{\text{rand}}, r) \quad \triangleright \text{Draw a line segment of length } r
6: \text{if } \text{CollisionFree}(s_{\text{nearest}}, s_{\text{new}}) \text{ then}
7: \quad S_{\text{near}} \leftarrow \text{Near}(V, s_{\text{new}}, r)
8: \quad V \leftarrow V \cup \{s_{\text{new}}\}
9: \quad s_{\text{min}} \leftarrow s_{\text{nearest}}
10: \quad c_{\text{min}} \leftarrow \text{Cost}(s_{\text{nearest}}) + \text{Cost(Line}(s_{\text{nearest}}, s_{\text{new}}))
11: \text{for each } s_{\text{near}} \in X_{\text{near}} \text{ do} \quad \triangleright \text{Connect along a minimum-cost path}
12: \quad c_{\text{near}} \leftarrow \text{Cost}(s_{\text{near}}) + \text{Cost(Line}(s_{\text{near}}, s_{\text{new}}))
13: \quad \text{if } \text{CollisionFree}(s_{\text{near}}, s_{\text{new}}) \text{ and } c_{\text{near}} \leq c_{\text{min}} \text{ then}
14: \quad s_{\text{min}} \leftarrow s_{\text{near}}; \; c_{\text{min}} \leftarrow c_{\text{near}}
15: \quad E \leftarrow E \cup \{(s_{\text{min}}, s_{\text{new}})\}
16: \text{for each } s_{\text{near}} \in X_{\text{near}} \text{ do} \quad \triangleright \text{Rewire the tree}
17: \quad c_{\text{near,new}} \leftarrow \text{Cost}(s_{\text{new}}) + \text{Cost(Line}(s_{\text{new}}, s_{\text{near}}))
18: \quad \text{if } \text{CollisionFree}(s_{\text{near}}, s_{\text{new}}) \text{ and } c_{\text{near,new}} \leq c_{\text{near}} \text{ then}
19: \quad s_{\text{parent}} \leftarrow \text{Parent}(s_{\text{near}})
20: \quad E \leftarrow (E \setminus \{(s_{\text{parent}}, s_{\text{near}})\}) \cup \{(s_{\text{new}}, s_{\text{near}})\}
21: \text{return } G = (V, E)

3 Related Work

What is the main reason why memory-based reasoning over actually-observed transitions is necessary? Why are methods like SPTM or SoRB that solely rely on learning-based distance estimates inherently prone to false predictions? By definition, states that are far apart from each other in terms of physical reachability are rarely observed together in an experiential learning framework (e.g., within the same RL episode, or within close by time-steps during random exploration). Therefore, for any learning-based prediction model that is conditioned on current-goal state pairs (e.g., \( Q(s_t, a, s_g) \), \( \pi_{\text{inv}}(a' | s_t, s_g) \), \( p_t(T' | s_t, s_g) \)), if it is trained solely on the observed distribution of experiential data, far apart states will be out of distribution (i.e., they have a low probability of being sampled from the experiential data distribution, therefore they are underrepresented in the replay buffer). This means that predictions for such far apart states will inevitably be inaccurate, unless a hard-negative sampling mechanism is implemented to explicitly push their reachability-estimates lower. The problem with this is that there is no inherent signal solely contained in perceptual input (e.g., images) that can guide the resampling process in a way that is generally applicable to all tasks.
While employing an ensemble of Q-functions could potentially capture the epistemic uncertainty for the baselines were chosen to facilitate a clear understanding of our approach without confounders, our SPTM:

$\text{SoRB}$

SoRB:

For example, \cite{53} employs a hard-negative sampling mechanism for a manipulation task using joint pose labels for guidance, but such a mechanism has two bottlenecks: i) it is specific to their particular manipulation task, ii) it assumes auxiliary labels. Similarly, the temporally consistent localization mechanism used in SPTM is essentially a heuristic fix specific to navigation. For the case of SoRB, there is no inherent mechanism in distributional RL that addresses this out-of-distribution issue either.

What are the details for our implementations of SoRB and SPTM: The main difference of our SPTM implementation is that it doesn’t employ temporally consistent localization and adaptive waypoint selection. The main differences of our SoRB implementation are: i) we use an ensemble of Q-functions, but they are trained with DDQN rather than distributional RL, ii) we train the Q-function on offline random-walk data, rather than an online episodic training setup with resets and a reward oracle as employed in the original paper. We acknowledge and emphasize that for SoRB, these differences are the most likely reason for the lower performance level we observed in our evaluations compared to the original paper, as they inevitably reduce the accuracy of Q-values. We however note that our method also employs the same Q-values, and generally these implementation differences in the baselines were chosen to facilitate a clear understanding of our approach without confounders.
Algorithm 11 PALMER: Perception-Action Loop with Memory Retrieval

1: Input: \(\mathcal{R}(\tau)\) 
2: \(f_\phi, \text{init}(), Q, \text{init}(), M \leftarrow \emptyset\) \hspace{1cm} \triangleright \text{Initialize policy parameters}
3: \textbf{while } t \leq \text{max\_timestep} \textbf{ do} 
4: \hspace{1cm} \textbf{while } i \leq \text{num\_exploration\_steps} \textbf{ do} \hspace{1cm} \triangleright \text{Exploration}
5: \hspace{2cm} i) Using [\text{any suitable method}]: explore the environment to obtain an exploration trajectory \(\tau_{\text{new}}\)
6: \hspace{2cm} ii) Using \(\tau_{\text{new}}\): \text{update} \(M\) \hspace{1cm} \triangleright \text{Memory expansion}
7: \hspace{1cm} \textbf{while } i \leq \text{num\_updates} \textbf{ do} 
8: \hspace{2.5cm} Using \([M]\): \text{update} \(Q(s_t, a_t, s_g)\) \hspace{1cm} \triangleright \text{Train value function}
9: \hspace{2.5cm} Using \([M]\ and \ Q(s_t, a_t, s_g)\): \text{update} \(f_\phi(s_t, s_g)\) \hspace{1cm} \triangleright \text{Train perception model}
10: \hspace{1cm} \textbf{while } i \leq \text{num\_exploitation\_steps} \textbf{ do} \hspace{1cm} \triangleright \text{Exploitation}
11: \hspace{2cm} i) Using \([M]\): sample a random goal \(s_g \sim M\)
12: \hspace{2cm} ii) Using \([f_\phi(s_c, s_g)\ and\ \mathcal{R}(\tau)]\): generate \(\tau_{M^*(s_c, s_g)}\)
13: \hspace{2cm} iii) Using [\text{any suitable method}]: execute \(\tau_{M^*(s_c, s_g)}\) to obtain a real trajectory \(\tau_{\text{real}}\)
14: \hspace{2cm} iv) Using \(\tau_{\text{real}}\): \text{update} \(M\) \hspace{1cm} \triangleright \text{Memory optimization}

because: i) they do not directly address the root cause of the false prediction problem (as discussed above), ii) one of the main benefits of our method is that it operates over arbitrary offline data without any resets or reward oracles (and it uses DDQN to train the related Q-function).

4 Experiments

Setup

![Visualization of the map used in VizDoom experiments.](image)

**Figure 12:** Visualization of the map used in VizDoom experiments. Further video visuals of image-based navigation can be found in the folder vizdoom\_nav provided in the supplementary alongside this document.

What are the details for the experimental setup in VizDoom: As shown in Fig.12, states solely consist of four images \(I_{\text{North}}/E\text{ast}/S\text{outh}/W\text{est}\) that form a panorama (i.e., \(4 \times 3 \times 160 \times 120\) dimensions), and actions move the agent North/South/East/West by a fixed distance \(\Delta\). The geodesic distances scale approximately by a factor of \(\times 3\) compared to euclidean distances (e.g., If a goal has an euclidian distance of \(14\Delta\), it takes approximately \(42\) timesteps for an optimal policy to reach it). The map contains many long-thin column-like obstructions (e.g., torches, pillars, trees), as we found that image-based navigation policies are prone to getting stuck in such obstacles. These obstacles have dynamically changing appearances (e.g., flickering flames on torches, glowing lights on pillars), and can completely block the field of view of the agent after a collision (as shown in \(I_{\text{East}}\) in Fig[12]. The replay buffer \(M\) consists of 300k images obtained from a uniform random walk exploring the map in a single continuous sequence of actions, without resets and rewards.
Validating Perceptual Experience Retrieval (PER)

What is the exact evaluation process that produced Fig. 4 in the main paper: For every integer value \( n \in [0, 14] \), we randomly sample 1000 pairs of start and goal-states in a way that the euclidian distance between them lies within \( n \times \Delta \) and \( (n + 1) \times \Delta \) through rejection sampling, and a policy is considered successful if it can get within \( \Delta \) proximity of the goal-state.

Robust Distances

What is the exact evaluation process for the right panel of Fig. 5 in the main paper: To produce the roadmap visualizations, we randomly sample 250 states from the replay buffer and set the edges between them by thresholding the distance estimates from all methods. Thresholds were calibrated individually and by hand for each baseline, by picking the threshold with the lowest number of false edges until a further reduction in the threshold resulted in splitting the roadmap into a large number of isolated subgraphs (i.e., therefore making it impossible to use it for global planning).

Proposed Planning Algorithms

Why does the policy \( \pi_{M^*} \) use R-PRM for planning: PRM and R-PRM are multi-query methods \([25]\), meaning that the full roadmap only needs to be constructed once. For any query pair of current-goal states \((s_c, s_g)\), the same roadmap can be reutilized by inserting \((s_c, s_g)\) in the roadmap and performing a shortest path query. In contrast, RRT and RRT* require recreating a full roadmap for every new query pair. Since \( \pi_{M^*} \) replans at each timestep with a different current state \( s_t \), PRM based approaches are computationally much cheaper. We also note that planning graphs for all methods in this experiment contain 500 vertices.

What are the details for the \( \pi_{mpc} \) baseline: The \( \pi_{mpc} \) policy uses the \( p_{fwd} \) model to obtain simulated rollouts, and uses the \( p_t \) model to rank those rollouts in terms of how close they get to the goal. This allows an MPC optimization loop that picks and implements the first action from the most successful simulated rollout. As previously discussed above, the main bottleneck for SPTM and SoRB is the difficulty of estimating reachability metrics solely using the two states \( s_c, s_g \) without any consideration of the states in between. Simulated rollouts in \( \pi_{mpc} \) naturally address this problem by generating and evaluating entire trajectories. The main bottleneck for \( \pi_{mpc} \) is that the accuracy of state predictions in simulated rollouts from \( p_{fwd} \) degrade with the rollout length.

Experiments in Habitat

![Figure 13: Visualization of the apartments used in Habitat experiments. Further video visuals of image-based navigation can be found in the folders habitat_roxbox_nav and habitat_annawan_nav provided in the supplementary alongside this document.](image)

What are the details for the experimental setup in Habitat: As shown in Fig. 13, states solely consist of a single 150 FOV image (i.e., \( 3 \times 256 \times 256 \) dimensions). There are 3 actions: \{turn_left_\_30_deg, turn_right_\_30_deg, move_for\_forward_\_\Delta\}. We run evaluations on two randomly picked apartments: Roxbox, and Annawan. In both apartments, we collect a replay buffer \( \mathcal{M} \) that consists of 150k images obtained from a uniform random walk exploring the map in a single
continuous sequence of actions, without resets and rewards. We use only the memory buffer from Roxbox to train the perception model \( f_p \), and use this same model to do perceptual experience retrieval and trajectory stitching on replay buffers from both apartments. We have observed that the latent distances from \( f_p \) generalize well, and can directly allow perceptual experience retrieval and trajectory stitching without any fine-tuning on the images from the new test apartment.

Why does the agent occasionally take random-looking actions in the habitat navigation trials? This is due to a combination of two main factors. First, our MPC loop replans from scratch at each timestep using Algorithm 6. This frequent replanning has a destabilizing effect on the control loop, similar to employing a large derivative action in a PID controller (i.e., a strong anticipatory term causes frequent switches in the actions). This first factor is exacerbated by the second main factor: the poor performance of \( \text{argmax}_a Q(s_t, a, s_g) \). This is most likely due to the difficulty of offline RL training with hindsight relabelling over random-walk data of only 150k timesteps obtained with a much more challenging non-cartesian action space \{ turn_left_30_deg, turn_right_30_deg, move_forward_\Delta \}. The restitched trajectories \( \tau_{M^*(s_t, s_g), s_1} \) produced by R-PRM at each time-step are converted to actions by following their first state \( \tau_{M^*(s_t, s_g), s_1} \) using \( \text{argmax}_a Q(s_t, a, \tau_{M^*(s_t, s_g), s_1}) \), and inaccurate Q-values occasionally cause random-looking actions.

There are multiple ways to counter this. The most direct way is to train a better Q-function. Our current Q-function is trained in a particularly challenging setting, as: i) it is trained on entirely offline data with hindsight goal relabelling, ii) this data is collected through uniformly random actions, and iii) it consists of only 150k environment steps. A second way is to instead counter the large derivative action by reducing the replanning rate and introducing a momentum mechanism to the controller. For example, we can replan through R-PRM only every \( n \)’th timestep (i.e., hence reducing the replanning rate), and act according to \( \text{argmax}_a Q(s_t, a, \tau_{M^*(s_t, s_g), s,n}) \) for the timesteps inbetween (i.e., hence introducing momentum to the controls). We couldn’t get this fix to work well, because the Q-values are only accurate up to states that are \( \sim 2 \times \Delta \) distance away (hence acting according to \( \text{argmax}_a Q(s_t, a, \tau_{M^*(s_t, s_g), s,n}) \) isn’t possible for \( n \geq 2 \) in our case). We note however that our method can still navigate from any point to any point in a challenging 3D reconstruction of a real-world apartment in Habitat using poor Q-value estimates, highlighting the robustness introduced by memory-based planning.

5 Discussion and Future Directions

How is PALMER related to the “Options Framework (Sutton et al.)” and “Skill-Chaining (Konidaris et al.)”? The idea of restitching (i.e., chaining) transition sequences from a replay buffer has direct connections to the options framework [55] and skill-chaining [56]. Essentially, PALMER can be thought of as a framework for converting every possible sequence of transitions \( \tau \in \mathcal{M} \) in memory into an option \( o = \{ \pi_o, I_o, \beta_o \} \), where the option policy \( \pi_o \) is implemented by simply executing all the actions in \( \tau \) in an open-loop manner, the initiation set is \( I_o = \{ s \in \mathcal{S} : d_\phi(s, \tau_{s,0}) \leq d_p \} \), and the termination condition is \( \beta_o = \{ s \in \mathcal{S} : d_\phi(s, \tau_{s,-1}) \leq d_p \} \). Therefore, PALMER can be interpreted as a skill-chaining algorithm that converts unstructured transitions in a replay buffer into a set of executable options to be chained.

How is PALMER related to “LQR-Trees (Tedrake et al.)”? A central idea in PALMER is repurposing the edge creation subroutines of sampling-based planning algorithms so that whenever an edge is created some additional processing is done to connect the endpoints (i.e., particularly, perceptual experience retrieval in our case). This approach is directly inspired by the method of LQR-Trees [57], which instead creates a trajectory stabilizing LQR controller to connect the endpoints of each edge. This results in a roadmap of local controllers, rather then a roadmap of memories as in PALMER.