Generative Proto-Sequence: Sequence-Level Decision Making for Long-Horizon Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

Deep reinforcement learning (DRL) methods often face challenges in environments characterized by large state spaces, long action horizons, and sparse rewards, where effective exploration and credit assignment are critical. We introduce Generative Proto-Sequence (GPS), a novel generative DRL approach that produces variable-length discrete action sequences. By generating entire action sequences in a single decision rather than selecting individual actions at each timestep, GPS reduces the temporal decision bottleneck that impedes learning in long-horizon tasks. This sequence-level abstraction provides three key advantages: (1) it facilitates more effective credit assignment by directly connecting state observations with the outcomes of complete behavioral patterns; (2) by committing to coherent multi-step strategies, our approach facilitates better exploration of the state space; and (3) it promotes better generalization by learning macro-behaviors that transfer across similar situations rather than memorizing state-specific responses. Extensive evaluations on mazes of varying sizes and complexities demonstrate that GPS consistently outperforms leading action repetition and temporal methods, where it converges faster and achieves higher success rates across all environments.

1 Introduction

Deep reinforcement learning (DRL) has demonstrated impressive performance across diverse applications (Mnih et al., 2015; Silver et al., 2016; Levine et al., 2016). However, significant challenges remain when dealing with environments characterized by large state spaces, long-range tasks, and sparse rewards. In such contexts, traditional DRL methods that select actions sequentially often suffer from inefficient exploration and poor credit assignment (Mesnard et al., 2020; Raileanu & Rocktäschel, 2020; Ecoffet et al., 2021), leading to difficulties in learning effective policies for tasks that require coordinated, multi-step strategies. These challenges are further exacerbated by sparse reward signals, whose limited feedback hinders the agent's ability to discover and reinforce successful behaviors (Arjona-Medina et al., 2019; Hung et al., 2019).

Recent research efforts have attempted to address these challenges using diverse techniques such as hierarchical DRL (Kulkarni et al., 2016; Xu et al., 2022), temporal abstraction (Biedenkapp et al., 2021; Zhang et al., 2022b; Saanum et al., 2023; Patel & Siegelmann, 2024), sequence modeling (Chen et al., 2021; Janner et al., 2021; Giladi & Katz, 2023), and action repetition strategies (Srinivas et al., 2017; Sharma et al., 2017; Dabney et al., 2020). By creating sub-tasks or capturing higher-level behavioral patterns (Rosete-Beas et al., 2023; Vezzani et al., 2022; Wang et al., 2023), these techniques aim to reduce decision frequency and enhance learning efficiency in long-horizon tasks. Although these methods offer promising results, they often require careful sub-task design (Ajay et al., 2023), reward shaping (Liu et al., 2022), or complex training procedures (Seo & Abbeel, 2024b; Rosete-Beas et al., 2023). While there are temporal abstraction methods that generate multi-step action sequences, they often depend on iterative rollouts, autoregressive decoding, or model-based simulations. These solutions introduce computational overhead and restrict the ability to efficiently generate diverse action sequences. To our knowledge, no method supports the generation of *coherent, variable-length action sequences directly from state observations in a single decision step.*

In this study, we propose Generative Proto-Sequence (GPS), a novel actor-critic architecture capable of producing variable-length action sequences. Instead of actions, our Actor generates a *proto-sequence* embedding, which is then decoded into a discrete action sequence using a Decoder component. The Critic evaluates the state and the entire generated sequence jointly, with gradients flowing from the critic through the Decoder to the Actor, facilitating end-to-end learning of strategic, multi-step action sequences. This design enables the agent to generate and execute complex exploratory behaviors in a single decision, enhancing both generalization and long-horizon credit assignment.

We evaluated GPS on a large set of challenging maze environments with varying sizes and configurations, including rooms, corridors, and randomly generated obstacles. Our results demonstrate that GPS consistently learns more efficiently, generalizes better to novel maze layouts, and significantly outperforms leading baselines in terms of success rate and convergence speed, particularly in large and complex mazes. Our contributions are as follows:

- We introduce a novel architecture that enables end-to-end generation and evaluation of variable-length discrete action sequences, facilitating improved credit assignment and exploration.
- We demonstrate that producing multi-step action sequences in a single shot leads to superior generalization and faster convergence, particularly in large and complex environments.
- We provide extensive empirical results on challenging maze benchmarks, showing significant improvements over top-performing action repetition and temporal methods baselines in metrics such as convergence speed and success rate.

2 Related Work

2.1 Temporal Abstraction Through Action Repetition

Early works in temporal abstraction explored repeating single primitive actions to extend decision horizons. Recent research in DRL has produced various approaches for performing multiple actions as a single block. Earlier works (Srinivas et al., 2017; Sharma et al., 2017) introduced frameworks for dynamic action selection and repetition, though their repetition policies operated independently from chosen actions, limiting strategic development.

DAR (Srinivas et al., 2017) augments discrete action spaces by duplicating each base action with multiple repetition rates. While this expansion can improve learning in environments benefiting from temporal abstraction, it produces an inefficient representation—duplicated actions are treated as unrelated, preventing the agent from exploiting their shared underlying behavior and leading to slower learning and imbalanced trade-offs between coarse and fine control. FiGAR (Sharma et al., 2017) addresses this by decoupling behavior and repetition into two jointly trained policies; however, the repetition policy operates independently from the chosen action, limiting the development of nuanced, action-specific repetition strategies.

The authors of (Dabney et al., 2020) proposed an exploration strategy repeating actions for random durations to reduce inefficient dithering. Temporl (Biedenkapp et al., 2021) advanced this by enabling agents to determine both action and repetition duration, improving learning efficiency. However, its hierarchical structure artificially decouples action selection from duration determination. Despite showing promise, these studies share a limitation: *temporal abstraction is achieved solely through simple repetition of primitive actions*, without generating coherent, variable-length action sequences.

2.2 Multi-Step Action Sequence Generation

Beyond single-action repetition, several methods focus on generating and partially committing to multi-step action sequences. The authors of (Zhang et al., 2022a) introduced a generative planning method (GPM) that produces multi-step plans. Since GPM is trained by maximizing value, the plans generated from it can be regarded as intentional action sequences to reach high-value states and improve sample efficiency. PrAC (Coad et al., 2022) enables agents to generate n-step plans and commit to them while being predictable, balancing adaptability and control stability. The work of (Saanum et al., 2023) incentivizes compressible action sequences by integrating sequence priors, while (Patel & Siegelmann, 2024) introduced a model-based sequence RL framework (SRL) reducing decision frequency through action chunking.

Despite recent progress, most existing methods for generating multi-step action sequences still face major limitations. Many rely on heavy processes such as iterative rollouts, autoregressive decoding, or model-based simulation, which can be slow and inflexible (Li et al., 2024; Li, 2023; Zhang et al., 2025). For example, methods like PrAC and SRL use learned environment models for both planning and training, adding extra model-based complexity (Kumar et al., 2024; Luo et al., 2024). To stay adaptable, some approaches also use external switching mechanisms or mid-sequence re-planning, as seen in GPM and PrAC. This treats long-term planning as an add-on to a step-by-step framework rather than as a core design principle. As a result, sequence generation and evaluation are often optimized separately, which can lead to poor credit assignment (Dai et al., 2018). One case is the use of handcrafted regularization, such as rewarding shorter or more "compressible" sequences (Saanum et al., 2023). However, when objectives are split in this way, it becomes unclear whether failures come from a bad plan or from breaking the secondary constraint, making end-to-end training harder and reducing stability during execution.

2.3 Temporal Abstraction Using Hierarchies and LLMs

Hierarchical methods have advanced multi-action decision-making through skill discovery and sequencing. TACO-RL (Rosete-Beas et al., 2023) learns latent skills from unstructured data for long-horizon tasks. ASPiRe (Xu et al., 2022) accelerated RL by combining specialized skill priors. The work of (Vezzani et al., 2022) introduced a skill scheduler sequencing pretrained skills, while SHRL (Wang et al., 2023) combined high-level policies with low-level skills for visual navigation. These approaches improve temporal abstraction by leveraging reusable skills rather than primitive actions.

Recent works have leveraged large language models and value-based reinforcement learning methods with action discretization for action sequence generation. CQN-AS (Seo & Abbeel, 2024a) proposed a value-based algorithm learning precise value functions from noisy action sequences. AlphaMaze (Dao & Vu, 2025) improved LLMs' spatial reasoning by combining supervised fine-tuning with policy optimization.

Our work draws inspiration from (Dulac-Arnold et al., 2015), who generated embedding representations of proto-sequences mapped to discrete actions. In GPS, we propose key improvements: our approach is fully differentiable and trainable end-to-end, unlike (Dulac-Arnold et al., 2015) whose k-nearest neighbors mapping broke the computation graph. Additionally, by using a VAE-based decoder instead of clustering, we automatically create sequence representations without manual embedding design. This enables efficient generation of coherent, variable-length action sequences that extend beyond simple repetition or skill sequencing.

3 Method

Overview. Our proposed approach is presented in Figure 1. GPS consists of three components: *Actor*, *Proto-Sequence Decoder (PSD)*, and *Critic*. The Actor receives the current state as input, and produces a *proto-sequence* – an embedding-based representation of a sequence of actions. The PSD receives the proto-sequence as input, and translates it into a discrete set of actions (e.g., $a_t, a_{t+1}, ..., a_{t+L}$), which are then executed sequentially by the agent. Finally, our Critic receives the sequence and predicts the expected cumulative reward obtained from its execution.

GPS differs from previous studies in several important aspects. First, unlike previous studies (Dulac-Arnold et al., 2015), it is end-to-end differentiable and does not require training workarounds. Secondly, our VAE decoder produces more diverse and flexible action sequences than autoregressive or model-based approaches, and also does so in an efficient, one-shot manner. Thirdly, sequence generation and evaluation are learned jointly, without regularization or switching mechanisms, thus improving credit assignment. Finally, by committing to the entire sequence (unlike the frequent re-evaluation of (Zhang et al., 2022a)) we reduce execution overhead and increase behavioral predictability by forcing GPS to learn robust policies.

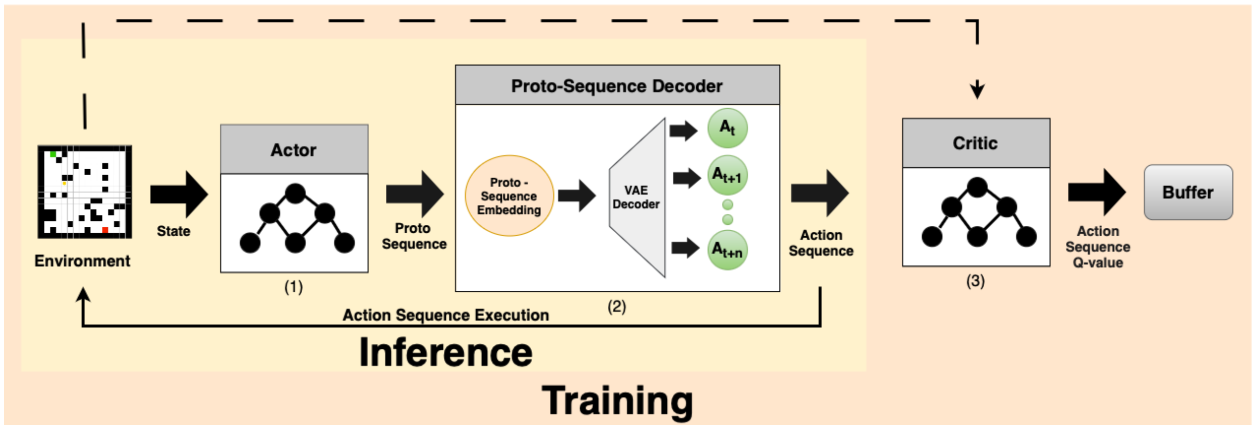

Figure 1: The three components of our proposed approach: (1) The Actor encodes the current state to produce a proto-sequence embedding. (2) The Decoder translates this latent embedding into a variable-length discrete action sequence. (3) The Critic evaluates the state-action-sequence pair and assigns it a Q-value representing the expected cumulative reward (3). During inference, only the actor and decoder components are used.

Another important aspect of our proposed approach is its ability to generate action sequences that differ from those on which it was trained. By creating novel sequences, GPS does not simply "memorize" a fixed set of actions, but is able to generalize to larger action spaces. We elaborate on GPS's capacity to produce novel sequences in Section 5.

3.1 The Actor

The Actor serves as GPS's policy network. Given a state s_t, the Actor analyzes the input and outputs a proto-sequence embedding $k = \pi_{\theta^\pi}(s_t)$, where θ^π and π are the parameters of the Actor's neural network and the current policy, respectively. The proto-sequence, represented in the embedding space $k \in K$, is the latent embedding of a sequence of actions, rather than a directly executable action. This representation provides our Actor with significant flexibility, as it can create action sequences of varying length using a fixed-size representation.

The proto-sequence is next used by the PSD to produce a discrete sequence of actions, and this sequence is evaluated by the Critic (Section 3.3). The parameters θ^π of the Actor are then updated using an actor-critic approach analogous to the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2016), leveraging the learning signal provided by the Critic. Specifically, the actor's parameters θ^π are adjusted to produce proto-sequence embeddings k that maximize the expected cumulative reward estimated by the critic, $Q_{\theta^Q}(s_t, \mathbf{a})$. This optimization is achieved by updating θ^π to minimize the negative Q-value provided by the critic $-Q_{\theta^Q}(s_t, g_{\theta^\omega}(\pi_{\theta^\pi}(s_t)))$, using backpropagated gradients from the output of the critic network Q_{θ^Q}. These gradients pass through the decoder network g_{θ^ω} and subsequently through the actor network π_{θ^π}, enabling the update of the latter's parameters θ^π.

3.2 The Proto-Sequence Decoder

The goal of the PSD is to translate the latent proto-sequence k generated by the Actor into a sequence of executable actions in the original action space $\{a_t, a_{t+1}, ..., a_{t+L}\} \in A$. We define the PSD as a function $g_{\theta^\omega} : K \to A'^{L_{max}}$, parameterized by θ^ω, where A' extends A with an EOS token to handle variable-length sequences within a fixed-length format, padding shorter sequences as needed. This function maps from the latent proto-sequence space K to sequences of fixed length L_{max}.

We use a Variational Autoencoder (VAE) (Kingma & Welling, 2013) as our PSD. We train the architecture on a diverse set of synthetic action sequences of varying lengths. For detailed information on the generation process of these sequences, see Appendix E.1. After training, we discard the VAE's encoder and retain only the learned decoder network g_{θ^ω}. The decoder is integrated into our agent architecture, transforming the Actor's latent proto-sequence embeddings into sequences of discrete actions. GPS will then execute the full sequence, without changes or early stopping. We chose VAE for its efficiency, ability to generate complete sequences in a single step, and its structured latent space that enables smooth interpolation and principled probabilistic modeling.

We pre-train the PSD and keep its parameters fixed while jointly training the actor and critic. This modular setup preserves generalization, improves efficiency by avoiding decoder retraining, and allows the decoder to be transferred across environments with the same action space. Furthermore, it separates sequence structure learning from environment-specific policy optimization.

3.3 The Critic

The goal of our Critic is similar to the role of the critic in an actor-critic architecture. The Critic receives the current state s_t and the one-hot encoded discrete actions sequence $\mathbf{A} = (a_t, a_{t+1}, \ldots, a_{t+L})$ produced by the PSD. It then attempts to predict $Q_{\theta^Q}(s_t, \mathbf{A})$, which represents the cumulative discounted reward obtained by executing $\mathbf{A}$ and following the policy after the end of the sequence:

$$Q_{\theta^Q}(s_t, \mathbf{A}) \approx \mathbb{E}_{\pi,P}\left[\sum_{k=0}^{L-1} \gamma^k r_{t+k} + \gamma^L V^\pi(s_{t+L})\right]$$

where V^π is the value function under policy π, θ^Q are the Critic's parameters, and $L = \mathit{eff\_len}(\mathbf{A})$ denotes the effective length of the action sequence $\mathbf{A}$.

The Critic's parameters θ^Q are updated by minimizing the Mean Squared Error (MSE) loss against a Temporal Difference (TD) target y_t:

$$L(\theta^Q) = \mathbb{E}_{(s_t,\mathbf{A},\text{rewards},s_{\text{next}})}\left[(Q(s_t, \mathbf{A}; \theta^Q) - y_t)^2\right]$$

The target y_t is constructed from the sum of discounted rewards $R_t(\mathbf{A})$ obtained by executing sequence $\mathbf{A}$, and the discounted value of the subsequent state $s_{t+\mathrm{L}}$, estimated using target Actor ($\text{Actor}_{\text{target}}$) and target Critic ($Q_{\text{target}}$) networks:

$$y_t = R_t(\mathbf{A}) + \gamma^L Q_{\text{target}}(s_{t+L}, \text{PSD}(\text{Actor}_{\text{target}}(s_{t+L})); \theta^{Q-})$$

This update mechanism, which relies on TD errors and target networks, is characteristic of many actor-critic algorithms, and shares similarities with methods such as DDPG (Lillicrap et al., 2016). While the Critic learns to accurately predict $Q(s_t, \mathbf{A}; \theta^Q)$, the Actor is trained to produce proto-sequences that, when decoded by the PSD, maximize this predicted Q-value.

3.4 Training Set Augmentation Using Sequence Subsets and Inference

To enhance learning efficiency and improve credit assignment, our training procedure leverages reward information from subsequences of each executed action sequence. For each sequence $\mathbf{A} = (a_t, \ldots, a_{t+L})$ of length L, we extract transitions corresponding to multiple contiguous subsequences $(a_i, \ldots, a_j)$ where $t \leq i < j < t + L$. For each such subsequence starting from an intermediate state s_i, we calculate the accumulated discounted reward obtained during its execution. This process effectively generates multiple learning samples of varying temporal lengths from a single interaction sequence, enriching the training data.

The subsequence extraction strategies for these state-subsequence-reward tuples, which we add to the replay buffer, include two primary approaches: (1) prefix extraction, which fixes the starting state while varying the end point, and (2) suffix extraction, which fixes the goal state while varying the starting point. This bidirectional approach diversifies the replay buffer with different time scales and enables the Critic to learn value estimates $Q_{\theta^Q}(s_i, (a_i, \ldots, a_j))$ for sequences of different lengths concurrently. As shown in our analysis in Section 5, these extraction strategies significantly accelerate learning and improve overall performance. It is important to note that during inference (test time), our architecture does not utilize the Critic component, since no training takes place. Instead, the Actor and PSD produce the action sequence, and the latter is executed in full.

Table 1: The setup and properties of the mazes used in the evaluation.

Environment	Dist. from start to goal	Train Set Size	Train Optimal Avg. Path	Val Set Size	Val Optimal Avg. Path	Test Set Size	Test Optimal Avg. Path
8x8	[1 - 14]	100	5.14	100	5.49	1000	5.31
16x16	[16 - 26]	100	18.04	100	18.0	1000	17.98
16x16_obstacles_15%	[20 - 30]	100	21.02	100	21.35	210	21.31
16x16_obstacles_25%	[20 - 30]	100	21.63	100	21.34	400	21.54
16x16_rooms	[20 - 30]	100	20.93	100	21.01	585	21.02
16x16_corridors	[10 - 30]	100	12.84	100	12.76	545	13.14
24x24	[20 - 30]	100	23.39	100	23.26	1000	23.56
24x24_obstacles_15%	[10 - 20]	100	15.04	100	14.58	1000	14.73
24x24_obstacles_25%	[10 - 20]	100	15.8	100	15.05	1000	15.11

4 Experiments and Results

4.1 Evaluation Environment

Mazes are a foundational benchmark in DRL research, commonly used to evaluate an agent's ability to perform complex sequential decision-making and navigation tasks. Their structured yet variable environments provide a controlled setting for evaluating generalization, exploration, and memory, which are central to DRL performance (Pašukonis et al., 2023). We use four types of mazes in our evaluation:

- **Empty.** These mazes have no walls or obstacles, except for their boundaries.
- **Sparse Obstacles.** This setup has randomly placed obstacles in K% of the cells of each maze (e.g., 15%).
- **Rooms.** This setup consists of four large rooms with small doors between them. We also add randomly placed obstacles in 5% of open cells.
- **Corridors.** These mazes have only narrow corridors for the agent to navigate.

Similarly to (Dao & Vu, 2025), we use an LLM to produce the code used in our maze generation. Our code, as well as the mazes generated for our evaluation, are available in the appendix. All information on our generated mazes is presented in Table 1. For each maze size and type, the table presents: *a)* the sizes of our training, validation, and test sets, *b)* the range for the distance between the start and goal positions, and *c)* the average length of the optimal path.

4.2 Baselines & Evaluated Methods

We evaluate two versions of GPS and three discrete-action baselines: *DQN*, *TempoRL*, and *DAR*. Full implementation details of our approach are included in the Appendix.

GPS: Our primary approach generates action sequences using a VAE-based decoder with Gumbel-Softmax sampling. This stochastic mechanism applies a temperature-controlled softmax to produce action distributions that maintain differentiability while approximating discrete samples. The Gumbel-Softmax technique creates a relaxation of categorical distributions that preserves gradients for backpropagation, facilitating end-to-end training of our actor-critic architecture.

GPS-D: A deterministic variant of our approach that uses argmax operations with a straight-through estimator in the decoder instead of Gumbel-Softmax sampling. This version produces consistent, deterministic action sequences for each proto-sequence embedding.

DQN: Deep Q-Network (Mnih et al., 2013) is a foundational model-free DRL algorithm that learns state-action values. DQN utilizes experience replay and a target network to stabilize its learning.

DAR: Dynamic Action Repetition (Srinivas et al., 2017) extends discrete action spaces by repeating original actions at varying rates. DAR enables the agent to select different levels of temporal control, allowing for some action abstraction.

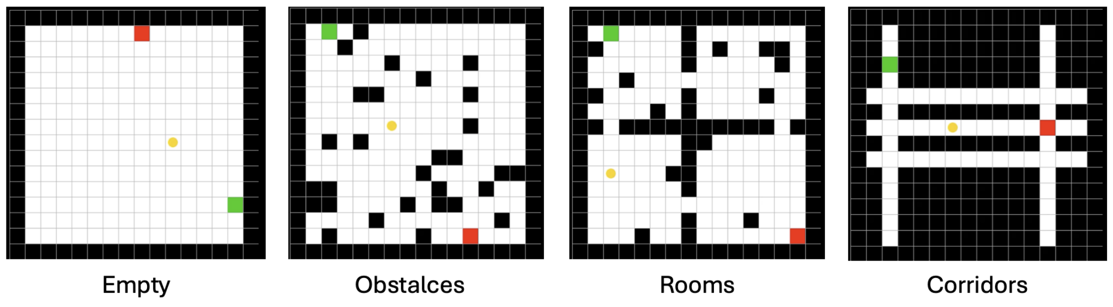

Figure 2: Examples of our generated mazes (16×16). We use four maze environments (left to right): **EMPTY** – open space; **15% Obstacles** – random obstacle placement; **ROOMS** – structured rooms with doorways; **CORRIDORS** – narrow paths requiring precise navigation.

TempoRL: Temporal Reinforcement Learning (Biedenkapp et al., 2021) introduces a proactive approach, where the agent selects both an action and its duration. TempoRL employs a hierarchical structure with a behavior policy for action selection and a skip policy for duration, enabling more fine-grained temporal abstraction and efficient exploration.

4.3 Experimental Setup

State and action representations. We represent the state using a tensor of shape $(N, M, 3)$, where N and M are the height and width of the maze grid. The channels use a similar encoding to that of MiniGridLibrary (Chevalier-Boisvert et al., 2023): *a) Object type*: identifies all environmental elements including walls, empty spaces, agent position, goal location, and starting position; *b) Object color*: provides distinguishing colors for the start position, goal location, and current agent position; *c) Placeholder channel*: consistently set to 0, maintaining compatibility with the MiniGrid format. Our discrete environment supports the four basic actions – up, down, left, right – represented as a four-entry one-hot vector

Reward function. We define the rewards function as follows:

$$R = r_{goal} - \frac{1}{l_{max}} \times n_{valid} - \frac{3}{l_{max}} \times n_{invalid}$$

where r_{goal} is 1 if the agent reached the goal (0 otherwise), l_{max} is the maximal start-goal distance (see Table 1) acting as a regularizer, n_{valid} is the number of valid actions taken, and $n_{invalid}$ is the number of invalid actions (e.g., bumping into a wall).

Evaluation metrics. We use three evaluation metrics:

- **Average success rate (ASR):** the percentage of episodes evaluated where the agent navigates successfully from start to goal position within a predefined number of steps.
- **Path efficiency ratio (PER):** for successfully completed episodes, we calculate the ratio between the episode length and the optimal (minimal) length:

$$PER = \frac{l_{opt}}{l_{episode}}$$

- **Sequence Generation Frequency (*SGF*):** This metric reflects how often the agent generates a new action sequence. It is calculated as the average number of times the Actor is invoked per evaluation episode. Lower values suggest the agent relies on longer-term proto-sequences before needing to generate a new sequence. This metric is relevant to GPS, DAR, and TempoRL baselines.

Table 2: ASR Performance at Different Training Steps

Environment	100k Steps					500k Steps					1M Steps					1.5M Steps				
	DQN	GPS	GPS-D	TempoRL	DAR	DQN	GPS	GPS-D	TempoRL	DAR	DQN	GPS	GPS-D	TempoRL	DAR	DQN	GPS	GPS-D	TempoRL	DAR
8x8	0.86	1.00	0.99	0.92	0.63	0.95	1.00	1.00	0.98	0.75	0.95	1.00	1.00	0.97	0.75	-	-	-	-	-
16x16	0.21	0.96	0.82	0.25	0.29	0.60	1.00	1.00	0.8	0.58	0.69	1.00	1.00	0.84	0.61	-	-	-	-	-
16x16_obst_15%	0.09	0.22	0.19	0.09	0.15	0.76	0.96	0.9	0.71	0.51	0.8	0.99	0.87	0.77	0.62	0.85	0.96	0.91	0.82	0.64
16x16_obst_25%	0.03	0.03	0.01	0.02	0.03	0.63	0.33	0.27	0.18	0.06	0.75	0.35	0.27	0.69	0.09	0.8	0.90	0.76	0.79	0.14
16x16_rooms	0.06	0.03	0.01	0.04	0.02	0.52	0.74	0.58	0.3	0.06	0.65	0.95	0.81	0.58	0.09	0.65	0.92	0.85	0.63	0.15
16x16_corr	0.4	0.98	0.9	0.61	0.15	0.81	1.00	0.96	0.9	0.53	0.8	1.00	0.97	0.9	0.61	-	-	-	-	-
24x24	0.04	0.14	0.1	0.01	0.05	0.14	1.00	0.94	0.28	0.15	0.24	1.00	0.99	0.46	0.23	-	-	-	-	-
24x24_obst_15%	0.02	0.12	0.06	0.02	0.04	0.08	0.45	0.24	0.05	0.06	0.11	0.80	0.49	0.16	0.10	0.15	0.91	0.56	0.20	0.12
24x24_obst_25%	0.03	0.06	0.02	0.02	0.03	0.08	0.30	0.17	0.03	0.03	0.11	0.36	0.17	0.04	0.04	0.11	0.36	0.16	0.09	0.07

Note: The ASR for each algorithm at specific training step intervals. A gray background indicates the highest ASR achieved for that environment across all steps and algorithms. A yellow background indicates the highest ASR within that specific step interval (excluding any cell already marked gray). '-' indicates unavailable data.

These metrics are complementary, as they allow us to evaluate the policy's *effectiveness*, *efficiency*, and *decision frequency*.

Neural architecture setup. All models use a shared CNN feature extractor followed by method-specific linear layers. DQN outputs Q-values for cardinal directions, DAR expands this for multiple repetition rates, and TempoRL implements a branching architecture for action and skip duration. In GPS, actor and critic networks use separate but identical CNN architectures. The actor produces a 16-dimensional proto-sequence embedding, which the decoder converts into action sequences through a multi-layer network with normalization. Full details are in the appendix.

Hyperparameters & Hardware. Unless otherwise noted for specific ablation studies, experiments were conducted using a common set of key hyperparameters, summarized in Tables 4–9 and 11 in the appendix. We selected the values based on preliminary experiments and common practices. All experiments were conducted on a system running Red Hat 5.14 with x86_64 architecture. We used an NVIDIA RTX 2080 GPU with 8GB of VRAM.

Training Protocol & Model Selection. We used different training setups based on maze size and type. Detailed step counts are in Table 2. Model selection for final testing used the checkpoint from each run yielding the highest average success rate on a held-out set of validation environments during training. Exploration employed an ϵ-greedy strategy, with random sequences being sampled from a predefined pool of 400 valid sequences (details in Appendix E.1).

4.4 Evaluation Results

4.4.1 Evaluating the Average Success Rate (ASR).

The results of our evaluation are presented in Table 2. GPS consistently outperforms the baselines in most cases, with several key observations:

Ability to learn, converge quickly, and generalize. GPS demonstrates high sample efficiency and rapid convergence. It achieved an ASR=0.96 on the 16×16 empty maze with only 100K training steps (compared to DQN's 0.21), and perfect accuracy (ASR 1.00) for 24×24 empty mazes at 500K steps, while the top baseline TempoRL only reached 0.28. This supports our hypothesis that modeling action sequences rather than individual actions enables more strategic exploration. GPS primarily learns to *generate sequences that move the agent in the correct general direction toward goals*, allowing progress in unseen environments even without perfectly optimized paths. The deterministic variant, GPS-D, also shows strong performance, supporting the robustness of the proto-sequence concept.

Scalability and superior ability to solve complex environments. The performance gap widens in larger environments. In the empty 24×24 maze, GPS achieves perfect performance at 500k steps, whereas DQN and TempoRL only reach 0.24 and 0.46 respectively after 1M steps. In complex 24×24 environments with 15% obstacles, our approach achieves an ASR=0.80 after 1M steps, almost eight times its closest

Table 3: Comparative Performance Analysis: Convergence Speed and Efficiency Metrics

Environment	ASR Converge>0.9 Step					PER					SGF				
	DQN	GPS	GPS-D	TempoRL	DAR	DQN	GPS	GPS-D	TempoRL	DAR	DQN	GPS	GPS-D	TempoRL	DAR
8x8	200k	100k	100k	100k	>1M	**1.0**	0.9	**1.0**	0.74	0.5	-	2.9	**2.6**	4.68	3.9
16x16	>1M	100k	200k	>1M	>1M	**1.0**	0.84	0.99	0.92	0.74	-	6.9	**5.1**	11.55	6.94
16x16_obstacles_15%	>1.5M	300k	700k	>1.5M	>1.5M	**1.0**	0.72	0.93	0.94	0.51	-	10.8	**9.2**	15.1	10.09
16x16_obstacles_25%	>1.5M	300k	700k	>1.5M	>1.5M	**1.0**	0.66	0.89	0.95	0.42	-	10.75	**9.12**	16.22	11.85
16x16_rooms	>1.5M	900k	>1.5M	>1.5M	>1.5M	**1.0**	0.63	0.89	0.95	0.63	-	13.76	**9.53**	15.01	9.84
16x16_corridors	>1M	100k	200k	500k	>1M	**0.99**	0.81	0.98	0.77	0.61	-	6.56	5.42	**4.77**	6.76
24x24	>1M	500k	600k	>1M	>1M	0.98	0.78	**1.00**	0.94	0.64	-	9.6	**7**	16.89	8.87
24x24_obstacles_15%	>1.5M	1.4M	>1.5M	>1.5M	>1.5M	**0.96**	0.48	0.84	0.93	0.45	-	**6.34**	**6.34**	13.34	7.76
24x24_obstacles_25%	>1.5M	>1.5M	>1.5M	>1.5M	>1.5M	**0.97**	0.48	0.79	0.91	0.41	-	14.9	8.66	13.87	**8.33**

Note: We present four key performance metrics. The first column shows training steps required to achieve a 90% success rate (lower is better), with highlighted values indicating the fastest convergence. Path Efficiency Ratio (PER) measures trajectory optimality (higher is better, max=1.0), with **bold values** showing best performance. Sequence Generation Frequency (SGF) indicates the average number of decision points needed per episode (lower generally indicates better temporal abstraction). '-' indicates N/A.

competitor. Even in the most difficult environments (24×24 with 25% obstacles), GPS maintains a significant relative advantage (ASR=0.36 vs. 0.14 for TempoRL).

Performance on medium-sized and structured environments. In the challenging "obstacles_25%" setup for 16×16 mazes, GPS outperforms DQN only after 1.5M training steps (ASR 0.90 vs 0.80). This is because medium-sized mazes do not significantly degrade DQN's performance, and dense obstacle distributions sometimes require very specific sequences. In structured "rooms" environments, GPS achieves an ASR of 0.95 at 1M steps, significantly outperforming DQN (0.65) and TempoRL (0.58). In the same 24×24 mazes, GPS performs far better, relatively.

Baselines like DAR and TempoRL generally struggle, particularly in complex mazes. For TempoRL, which might require more extensive training to converge optimally, we observed improved performance with larger training data (see Table 12 in the appendix), though computational constraints limited further exploration. GPS's strong performance stems from operating in the space of action sequences rather than individual actions, enabling more strategic exploration and the discovery of long-horizon rewards that would be difficult to find using single actions or simple repetition methods.

4.4.2 Evaluating the Path Efficiency Ratio (PER).

The results of our evaluation are presented in Table 3. PER is calculated at the final training checkpoint using the total time steps per environment detailed in Table 1. We report PER for GPS, GPS-D, and two baselines introduced in Section 4.2, allowing for direct comparison across methods. A key GPS characteristic is **Self-Correction Through Sequential Decision Points**. GPS can adjust its course at subsequent decision points without requiring an initially perfect action sequence. This sequence-level closed-loop control enables course corrections while retaining the benefits of temporal abstraction. Leveraging this capability, GPS adopts a strategy of **Trading Path Efficiency for Robust Navigation**, prioritizing directional correctness over strict path optimality. This approach develops more transferable navigation skills—particularly evident in larger or more obstacle-dense mazes—explaining cases where PER is lower despite higher ASR and faster convergence (see Tables 2 and 3).

GPS-D consistently yields higher PER than GPS in all environments. For example, in the 16×16 empty maze, GPS-D's PER is 0.99 versus GPS's 0.84; in the 24×24 maze with 15% obstacles, PER is 0.84 for GPS-D and 0.48 for GPS. GPS's Gumbel-Softmax sampling introduces stochasticity that enables broader exploration but can cause path deviations. GPS-D's deterministic *argmax* decoder produces more consistent trajectories, trading exploration advantages for improved exploitation. Among the baselines, *DQN* and *TempoRL* often show high PER, frequently achieving near-optimal paths. For instance, in 16×16 empty maze, *DQN* reached PER 1.0 and *TempoRL* 0.92. However, GPS often surpasses their ASR in complex environments. *DAR* generally shows a lower PER.

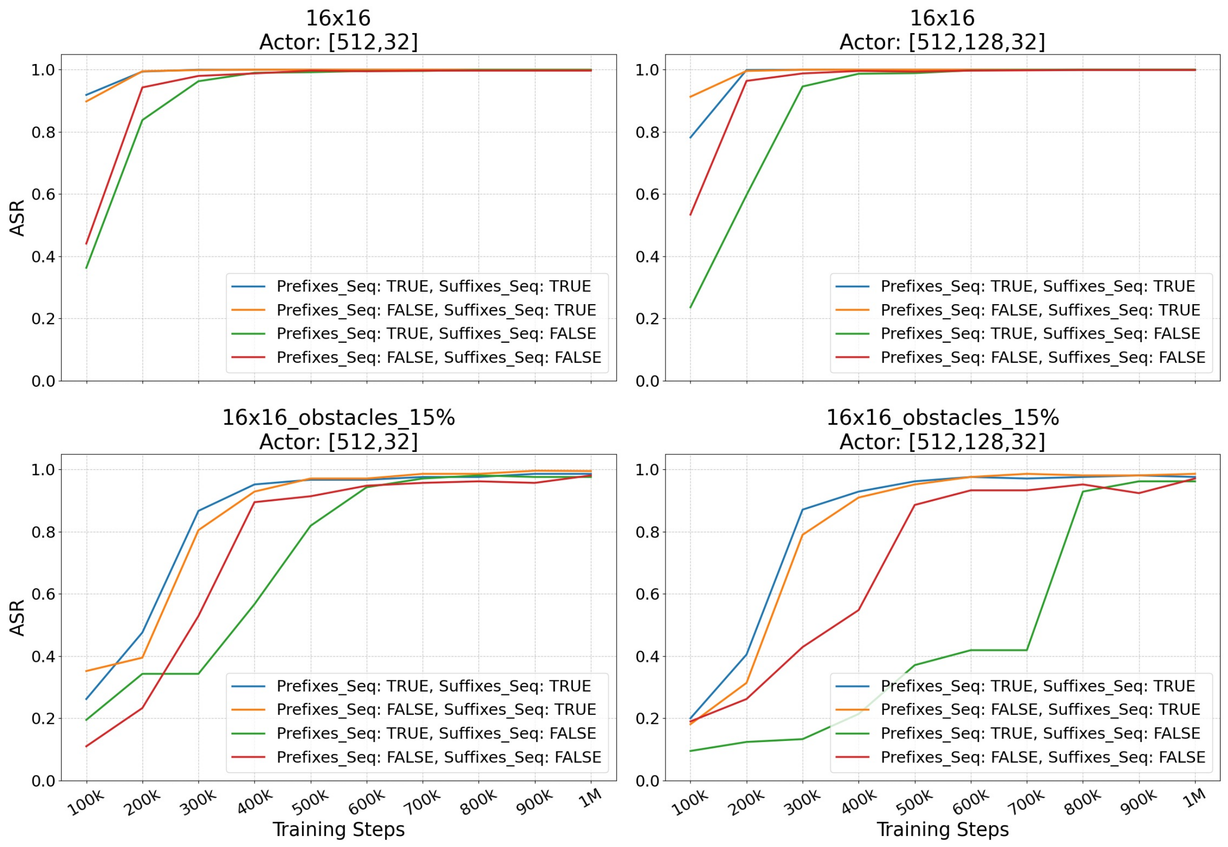

Figure 3: Impact of Subsequence Buffering Strategy on Average Success Rate.

In summary, while GPS may trade path optimality for higher ASR and faster learning, GPS-D shows excellent path efficiency. This highlights a trade-off: GPS's stochasticity boosts exploration and rapid ASR, while GPS-D's determinism excels in path efficiency once a good policy is learned.

4.4.3 Evaluating Sequence Generation Frequency (SGF).

The results are presented in Table 3, where lower values generally indicate superior temporal abstraction due to fewer policy invocations per episode. For methods reporting SGF, our approaches GPS and GPS-D demonstrate competitive performance across environments. In 16×16 empty maze, GPS-D achieves an SGF of 5.1, outperforming *DAR*'s 6.94 and *TempoRL*'s 11.55, while GPS records 6.9. However, in the 16×16 corridors environment, *TempoRL* (SGF 4.77) outperform GPS-D (5.42) and GPS (6.56). Despite this environment-dependent variation, our methods often operate with limited interventions-such as GPS-D's 7.0 SGF in 24×24 empty mazes versus *TempoRL*'s 18.1-demonstrating effective generation of extended proto-sequences.

GPS-D's generally low SGF combined with its high PER indicates capability for efficient, strategic trajectory generation through robust behavioral patterns, making it ideal for scenarios requiring predictable execution or constrained resources. GPS offers a compelling trade-off with competitive PER and favorable SGF compared to *TempoRL* (e.g., 6.9 vs. 11.55 in 16×16 empty; 9.53 vs. 15.01 in 16×16 rooms), alongside faster Average Success Rate convergence as discussed in Section 4.4.1. It balances path efficiency, sequence compactness, and learning speed effectively.

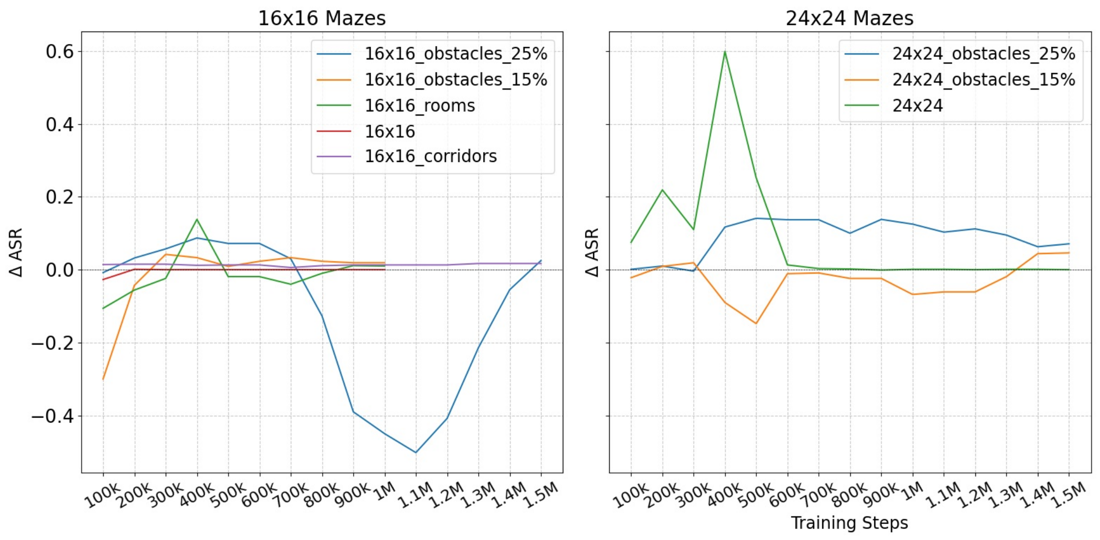

Figure 4: Impact of Actor Network Scaling on Average Success Rate (Δ ASR) in Mazes.

In conclusion, SGF analysis confirms our sequence-generation paradigm's effectiveness for temporal abstraction. GPS-D provides efficient, long-term utility with fewer, optimal decisions, while GPS balances competitive SGF, good PER, and rapid ASR. The choice between them depends on application priorities: efficiency vs. predictability or adaptation vs. broader performance.

5 Analysis and Discussion

Analyzing GPS's Ability to Generate Novel Sequences. We consider GPS's ability to generate novel action sequences – sequences that were not used in the training of the Decoder – to be a useful aspect of our approach. This capability enables our approach to diversify its generated sequences beyond those it was "taught" during training. Not being constrained by a fixed training set extends GPS's capabilities beyond memorization to generalization, as we show below.

As described in Appendix E.1, the PSD was pre-trained on a set of 400 synthetic sequences generated according to simple, common-sense heuristics for navigation tasks: *a)* each sequence contained at most two distinct action types. *b)* Actions of the same type appeared in contiguous blocks (e.g., "up, up, left" allowed; "up, left, up" disallowed). *c)* No immediately contrasting actions were allowed (e.g., "up, down" prohibited). *d)* Maximum sequence length was capped at $L_{\max}$ (shorter sequences permitted). *e)* Avoidance of loops. After full training, we gathered 15 action sequences by sampling states from the GPS replay buffer and generating the corresponding action sequences through the actor and PSD. Eleven of these did not appear in the PSD's training set and were not fully aligned with at least one of the navigation patterns described above. Using the encoding up→0, down→1, left→2, right→3, the novel sequences were:

$$\begin{gathered}[\,[1,2,1,1],\ [1,1,0],\ [3,1,0,1],\ [0,0,3,1],\\ [3,3,2],\ [3,0,3],\ [1,1,2,3],\ [2,0,1],\\ [0,0,3,3,0,3],\ [3,3,1,2],\ [2,1,1,3]]\end{gathered}$$

These results show that GPS can create new action sequences not seen during training because it works in a structured embedding space. In this space, sequences with similar structures are grouped together, making it possible to blend known patterns and generate new ones, as shown in Figure 5 in the appendix.

Sequence Subsets Augmentation. We investigate our subsequence buffering approach (Section 3.4), implemented through *prefixes* (fixed start, varying end point) and *suffixes* (fixed goal state, varying starting point). Figure 3 shows that the baseline without subsequence buffering (red) consistently learns most slowly and often converges sub-optimally, while all subsequence buffering variants substantially improve learning efficiency. Using prefixes and suffixes simultaneously (blue) generally produces the most rapid learning, though the suffix-only configuration (orange) performs nearly as well, suggesting backward sampling provides particularly valuable learning signals. The prefix-only approach (green) typically shows slower convergence than other subsequence methods. These performance patterns remain consistent across different maze structures and actor networks.

Impact of Actor Network Scaling. We examined actor network size impact (small: two-layer (512, 32); large: three-layer (512, 128, 32)) on maze navigation performance, measured by ΔASR (Large - Small) (see Figure 4). In simpler 16x16 mazes (empty or corridor), both architectures performed similarly. With 15% obstacles, the smaller network initially outperformed (ΔASR ≈ -0.3 at 100K steps) before convergence at 300K steps. In denser 25% obstacles, the smaller network significantly outperformed from 800K steps, peaking at ΔASR ≈ -0.5 at 1.1M steps. In larger 24x24 mazes with 15% obstacles, the small network generally led, despite the large network's brief advantage (500K steps). However, in the most complex 25% obstacles maze, the large network consistently outperformed, maintaining ΔASR between 0.1-0.15. The 24x24 empty maze showed fluctuating performance with occasional spikes for the larger network around 200K and 500K steps.

These results suggest a trade-off: smaller networks suffice or excel in smaller or moderately complex environments (possibly due to better regularization or more stable sequence generation learning), while larger networks demonstrate clear benefits in more complex environments.

6 Conclusions, Limitations, and Future Work

GPS is a novel actor-critic method that generates variable-length action sequences in a single step. GPS maps state observations to proto-sequences, which are decoded into discrete action sequences. This approach enhances credit assignment and exploration in long-horizon tasks by moving beyond sequential single-action selection. Our evaluation shows GPS consistently surpasses leading action repetition and temporal methods in complex maze environments, achieving higher success rates and faster convergence.

Although our approach shows benefits, particularly in complex environments, several limitations should be acknowledged. First, a new PSD needs to be trained for each unique action space, which adds to the complexity of our approach. GPS has not been evaluated on large action spaces, so adaptations to the decoder component may be needed. Additionally, we have not yet adapted GPS to continuous action spaces. Future work will focus on making GPS generalizable in more complex, realistic environments, explore reward function that incentivize path efficiency, and address continuous action spaces.

References

Anurag Ajay, Seungwook Han, Yilun Du, Shuang Li, Abhi Gupta, Tommi Jaakkola, Josh Tenenbaum, Leslie Kaelbling, Akash Srivastava, and Pulkit Agrawal. Compositional foundation models for hierarchical planning. *Advances in Neural Information Processing Systems*, 36:22304–22325, 2023.

Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

André Biedenkapp, Raghu Rajan, Frank Hutter, and Marius Lindauer. Temporl: Learning when to act. In *International Conference on Machine Learning*, pp. 914–924. PMLR, 2021.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Josiah D Coad, James Ault, Jeff Hykin, and Guni Sharon. A framework for predictable actor-critic control. In *Deep Reinforcement Learning Workshop NeurIPS 2022*, 2022.

Will Dabney, Georg Ostrovski, and André Barreto. Temporally-extended {\epsilon}-greedy exploration. *arXiv preprint arXiv:2006.01782*, 2020.

Zihang Dai, Qizhe Xie, and Eduard Hovy. From credit assignment to entropy regularization: Two new algorithms for neural sequence prediction. *arXiv preprint arXiv:1804.10974*, 2018.

Alan Dao and Dinh Bach Vu. Alphamaze: Enhancing large language models' spatial intelligence via grpo. *arXiv preprint arXiv:2502.14669*, 2025.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

Liad Giladi and Gilad Katz. Feedback decision transformer: Offline reinforcement learning with feedback. In *IEEE International Conference on Data Mining (ICDM)*. IEEE, 2023.

Chia-Chun Hung, Timothy Lillicrap, Josh Abramson, Yan Wu, Mehdi Mirza, Federico Carnevale, Arun Ahuja, and Greg Wayne. Optimizing agent behavior over long time scales by transporting value. *Nature communications*, 10(1):5223, 2019.

Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34:1273–1286, 2021.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

Nishanth Kumar, Tom Silver, Willie McClinton, Linfeng Zhao, Stephen Proulx, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Jennifer Barry. Practice makes perfect: Planning to learn skill parameter policies. *arXiv preprint arXiv:2402.15025*, 2024.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

Jinghan Li, Zhicheng Sun, and Yadong Mu. Closed-loop long-horizon robotic planning via equilibrium sequence modeling. *arXiv preprint arXiv:2410.01440*, 2024.

Wenhao Li. Efficient planning with latent diffusion. *arXiv preprint arXiv:2310.00311*, 2023.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations (ICLR)*, 2016.

Jinxin Liu, Donglin Wang, Qiangxing Tian, and Zhengyu Chen. Learn goal-conditioned policy with intrinsic motivation for deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pp. 7558–7566, 2022.

Fan-Ming Luo, Tian Xu, Hang Lai, Xiong-Hui Chen, Weinan Zhang, and Yang Yu. A survey on model-based reinforcement learning. *Science China Information Sciences*, 67(2):121101, 2024.

Thomas Mesnard, Théophane Weber, Fabio Viola, Shantanu Thakoor, Alaa Saade, Anna Harutyunyan, Will Dabney, Tom Stepleton, Nicolas Heess, Arthur Guez, et al. Counterfactual credit assignment in model-free reinforcement learning. *arXiv preprint arXiv:2011.09464*, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, and Georg Ostrovski. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Jurgis Pašukonis, Timothy P Lillicrap, and Danijar Hafner. Evaluating long-term memory in 3d mazes. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=yHLvIlE9RGN`.

Devdhar Patel and Hava Siegelmann. Overcoming slow decision frequencies in continuous control: Model-based sequence reinforcement learning for model-free control. *arXiv preprint arXiv:2410.08979*, 2024.

Roberta Raileanu and Tim Rocktäschel. Ride: Rewarding impact-driven exploration for procedurally-generated environments. *arXiv preprint arXiv:2002.12292*, 2020.

Erick Rosete-Beas, Oier Mees, Gabriel Kalweit, Joschka Boedecker, and Wolfram Burgard. Latent plans for task-agnostic offline reinforcement learning. In *Conference on Robot Learning*, pp. 1838–1849. PMLR, 2023.

Tankred Saanum, Noémi Éltető, Peter Dayan, Marcel Binz, and Eric Schulz. Reinforcement learning with simple sequence priors. *Advances in Neural Information Processing Systems*, 36:61985–62005, 2023.

Younggyo Seo and Pieter Abbeel. Coarse-to-fine q-network with action sequence for data-efficient robot learning. *arXiv preprint arXiv:2411.12155*, 2024a.

Younggyo Seo and Pieter Abbeel. Coarse-to-fine q-network with action sequence for data-efficient robot learning. *arXiv preprint arXiv:2411.12155*, 2024b.

Sahil Sharma, Aravind Srinivas, and Balaraman Ravindran. Learning to repeat: Fine grained action repetition for deep reinforcement learning. *arXiv preprint arXiv:1702.06054*, 2017.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Aravind Srinivas, Sahil Sharma, and Balaraman Ravindran. Dynamic action repetition for deep reinforcement learning. In *Proc. AAAI*, 2017.

Giulia Vezzani, Dhruva Tirumala, Markus Wulfmeier, Dushyant Rao, Abbas Abdolmaleki, Ben Moran, Tuomas Haarnoja, Jan Humplik, Roland Hafner, Michael Neunert, et al. Skills: Adaptive skill sequencing for efficient temporally-extended exploration. *arXiv preprint arXiv:2211.13743*, 2022.

Shuo Wang, Zhihao Wu, Xiaobo Hu, Youfang Lin, and Kai Lv. Skill-based hierarchical reinforcement learning for target visual navigation. *IEEE Transactions on Multimedia*, 25:8920–8932, 2023.

Mengda Xu, Manuela Veloso, and Shuran Song. Aspire: Adaptive skill priors for reinforcement learning. *Advances in Neural Information Processing Systems*, 35:38600–38613, 2022.

Dongkun Zhang, Jiaming Liang, Ke Guo, Sha Lu, Qi Wang, Rong Xiong, Zhenwei Miao, and Yue Wang. Carplanner: Consistent auto-regressive trajectory planning for large-scale reinforcement learning in autonomous driving. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 17239–17248, 2025.

Haichao Zhang, Wei Xu, and Haonan Yu. Generative planning for temporally coordinated exploration in reinforcement learning. In *International Conference on Learning Representations*, 2022a.

Haichao Zhang, Wei Xu, and Haonan Yu. Generative planning for temporally coordinated exploration in reinforcement learning. *arXiv preprint arXiv:2201.09765*, 2022b.

A Baselines and Architecture

Each baseline is evaluated using a grid search over multiple hyperparameter configurations; Tables 4, 5, 6, 7, 8, 9 and 11 detail the specific value ranges for these parameters.

All baseline models employ the same CNN feature extractor architecture followed by similarly sized linear layers, differing only in the final output layer size. For example, DQN outputs 4 Q-values (one per action), while DAR outputs 12 (4 actions $\times$ 3 repetition heads). TempoRL requires an additional network head to implement the skip policy, adding architectural complexity but gaining flexibility in temporal decision-making. In our GPS method, the actor and critic networks each have their own separate CNN state feature extractors. In future work, we plan to explore a shared CNN feature extraction architecture as implemented in TempoRL, which could potentially improve computational efficiency and state representation learning.

For TempoRL, we configured the model with a maximum skip length between 1..10 to allow variable sequence lengths of action repetition. For DAR, we evaluated possible coarse control values of 1,5,10 to allow the same maximum sequence length and mid-sequence capability, with the fine control value fixed at 1 to allow for actions at every time step. We based our implementations on the publicly available code at `https://github.com/automl/TempoRL` but reimplemented from scratch to enrich with more metrics and employ our evaluation methodology. Detailed architectures, hyperparameter configurations, and implementation specifics can be found in Appendix B, C, D, E and F.

B DQN Baseline Implementation Details

This section outlines the architecture and configuration of the Deep Q-Network (DQN) agent used as a baseline. It details the neural network structure, hyperparameter settings, exploration strategy, optimization method, and other relevant training aspects.

B.1 Model Architecture (QNetwork)

The Q-Network is a neural network designed to approximate the action-value function $Q(s, a)$. It consists of a convolutional part for feature extraction from the input observation and a linear part for producing Q-values for each action.

In our maze environments, as depicted in Subsection 4.3, the input observation has a shape (C, H, W), where the number of input channels C is 3. The number of output channels, $n\_output\_channels$, corresponds to the number of available actions, which is 4 (right, left, up, down).

B.1.1 Convolutional Neural Network (CNN) Part

The CNN component processes the input observation through a sequence of convolutional layers:

1. **Conv2D Layer 1**:
   - Input channels: 3
   - Output channels: 16
   - Kernel size: 2
   - Stride: 1
2. **Activation**: ReLU
3. **Conv2D Layer 2**:
   - Input channels: 16
   - Output channels: 32
   - Kernel size: 2
   - Stride: 1

4. **Activation**: ReLU
5. **Conv2D Layer 3**:
    - Input channels: 32
    - Output channels: 64
    - Kernel size: 2
    - Stride: 1
6. **Activation**: ReLU
7. **Flatten Layer**: The output of the convolutional layers is flattened into a 1D vector. The size of this vector, *n_flatten*, is computed automatically.

B.1.2 Linear Part

The flattened output from the CNN (*n_flatten*) is fed into a sequence of fully connected linear layers:

- The hidden layer sizes are configurable via grid search (see Table 4 for details). The activation function for these hidden layers is Leaky ReLU (negative slope 0.1).
- The final linear layer maps the last hidden layer's output to *n_output_channels* (4 actions).

B.2 Hyperparameters

The agent's behavior and training process are governed by a set of hyperparameters, detailed in Tables 4, 5, and 6.

Table 4: General Experiment Hyperparameters for DQN Baseline

Parameter	**Default Value**
`seed`	123
`torch_deterministic`	True
`save_model_strategy`	`SUCCESS_RATE`
`val_eval_freq`	5000
`train_eval_freq`	5000
`eval_test_dataset_training_freq`	100000

Table 5: Environment-Specific Hyperparameters for DQN Baseline

Parameter	**Default Value**
max_episode_steps	75
reward_strategy	`NEGATIVE_BASED_ON_MAX_LEVEL_WITH_PENALTIES`
observation_encoding_strategy	`DEFAULT`
Max Path Length (*max_level*)	Varies (see Env. Def. in Table 1)
Min Path Length (*start_level*)	Varies (see Env. Def. in Table 1)

B.3 Epsilon-Greedy Exploration

The agent uses an epsilon-greedy strategy for action selection. The value of epsilon (ϵ) is linearly annealed from `start_e` (1.0) to `end_e` (0.1) over a `duration`. This duration is calculated as $\lfloor$exp_frac $\times$ `total_timesteps`$\rfloor$, where exp_frac is the selected `exploration_fraction` (from options in Table 6) and t is the current global timestep. The epsilon at timestep t is:

$$\epsilon_t = \max((((\text{end\_e} - \text{start\_e})/\text{duration}) \times t + \text{start\_e}), \text{end\_e})$$

With probability ϵ_t, a random action is chosen; otherwise, the action with the highest Q-value is selected.

Table 6: Algorithm Specific Hyperparameters for DQN Baseline

Parameter	**Default Value / Options**
total_timesteps	Environment specific
learning_rate	$[1 \times 10^{-3}, 1 \times 10^{-4}]$
buffer_size	[10000, 50000]
γ (discount factor)	0.99
τ (target update rate)	[0.01, 0.005]
target_network_frequency	[10, 100] (soft-target update freq.)
batch_size	256
start_e	1.0 (initial ε)
end_e	0.1 (final ε)
exploration_fraction	[0.1, 0.3, 0.5]
learning_starts	1000 (timestep to begin learning)
train_frequency	2 (Q-network update freq.)
linear_layers	["512,128,32", "512,32"]
activation_function	Leaky ReLU (slope 0.1)

B.4 Optimizer

The Q-Network is trained using the Adam optimizer (`torch.optim.Adam`). The learning rate is controlled by the `learning_rate` hyperparameter (see Table 6).

B.5 Replay Buffer

A replay buffer (`stable_baselines3.common.buffers.ReplayBuffer`) stores experiences $(s_t, a_t, r_t, s_{t+1}, d_t)$.
The buffer size is specified in Table 6. Key configurations include `optimize_memory_usage = False` and `handle_timeout_termination = False`.

B.6 Training Details

Loss Function. The Q-Network parameters (θ) are updated by minimizing the Mean Squared Error (MSE) loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{B}}\left[(y_t - Q(s,a;\theta))^2\right]$$

where the TD target:

$$y_t = r_t + \gamma \max_{a'} Q_{target}(s_{t+1}, a'; \theta^-)(1 - d_t)$$

Here, r_t is the reward, γ is the discount factor, Q_{target} is the target network with parameters θ^-, and d_t indicates if s_{t+1} is terminal. This is implemented via `torch.nn.functional.mse_loss`.

Target Network. A separate target network Q_{target} with parameters θ^- stabilizes training. Its weights are updated using Polyak averaging: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$. The soft update rate τ and update frequency `target_network_frequency` are specified in Table 6.

Training Procedure.

- **Learning Starts**: Training begins after `learning starts` timesteps (see Table 6).
- **Training Frequency**: The Q-network is updated every `train_frequency` global steps (see Table 6).
- **Batch Size**: Number of experiences sampled per training step is `batch_size` (see Table 6).

B.7 Evaluation

The agent's performance is evaluated periodically on validation and test datasets.

- Evaluation on the validation dataset occurs every `val_eval_freq` steps.
- Evaluation on the test dataset can occur during training every `eval_test_dataset_training_freq` steps.
- During evaluation, actions are chosen greedily (or with a small fixed epsilon, e.g., 0.05 or 0.0).
- Metrics logged include mean episodic return, success rate, and agent step ratio.
- Model saving is based on performance metrics (e.g., highest success rate or reward on validation) as per `save_model_strategy`.

C DAR Baseline Implementation Details

This section outlines the architecture and configuration of the Dyanmic Action Repetition (DAR) agent used as a baseline. The DAR agent builds upon the Deep Q-Network (DQN) architecture and training methodology. Therefore, for aspects not explicitly mentioned here, such as the general experiment configuration (Table 4), environment-specific arguments (Table 5), epsilon-greedy exploration strategy (Section B.3), optimizer (Section B.4), replay buffer (Section B.5), general training procedure (Section B.6), and evaluation methodology (Section B.7), please refer to the corresponding descriptions in the DQN baseline implementation details (Section B).

The primary distinctions of the DAR baseline are its modified network architecture to support an expanded action space and an additional algorithm-specific hyperparameter, `dar_r_l`, related to action repetition.

C.1 Model Architecture

The DAR network for the DAR agent, similar to DQN, approximates the action-value function $Q(s, a)$. It comprises a convolutional part for feature extraction and a linear part for producing Q-values.

The input observation from the maze environments has a shape (C, H, W), where $C = 3$, identical to the DQN baseline (Section B.1).

C.1.1 Convolutional Neural Network (CNN) Part

The CNN component is identical to the one used in the DQN baseline. For details on the architecture (number of layers, channels, kernel sizes, strides, and activations), please refer to Section B.1. The output of this part is a flattened 1D vector of size $n\_flatten$.

C.1.2 Linear Part

The flattened output ($n\_flatten$) from the CNN is processed by a sequence of fully connected linear layers:

- The hidden layer sizes are configurable via grid search, with the same options as the DQN baseline (see Table 7 for `linear_layers`). The activation function for these hidden layers is Leaky ReLU (negative slope 0.1).
- The final linear layer maps the last hidden layer's output to $n\_output\_channels$. For the DAR agent, $n\_output\_channels = 12$, corresponding to 4 base actions (right, left, up, down) each associated with 3 repetition heads/levels.

C.2 Hyperparameters

The general experimental configuration and environment-specific hyperparameters for the DAR baseline are the same as those for the DQN baseline, as detailed in Table 4 and Table 5, respectively.

C.2.1 Algorithm Specific Arguments

The algorithm-specific hyperparameters for the DAR baseline, including the newly introduced `dar_r_l` parameter, are listed in Table 7. These parameters are subject to grid search to find the optimal configuration for each environment.

Table 7: Algorithm Specific Hyperparameters for DAR Baseline

Parameter	**Default Value / Options**
total_timesteps	Environment specific
learning_rate	$[1 \times 10^{-3}, 1 \times 10^{-4}]$
buffer_size	[10000, 50000]
γ (discount factor)	0.99
τ (target update rate)	[0.01, 0.005]
target_network_frequency	[10, 100] (soft-target update freq.)
batch_size	256
start_e	1.0 (initial ε)
end_e	0.1 (final ε)
exploration_fraction	[0.1, 0.3, 0.5]
learning_starts	1000 (timestep to begin learning)
train_frequency	2 (Q-network update freq.)
linear_layers	["512,128,32", "512,32"]
activation_function	Leaky ReLU (slope 0.1)
dar_r_l	[1, 5, 10] (repetition level parameter)

C.3 Training Details

Loss Function. For DAR, the Q-Network parameters (θ) are updated by minimizing Hubber loss. This is implemented via torch.nn.SmoothL1Loss.

D TempoRL Baseline Implementation Details

This section describes the architecture and configuration of the TempoRL agent, a baseline designed for temporal abstraction by learning how long to repeat actions. TempoRL shares several components and procedures with the DQN baseline. For details on the general experiment configuration (Table 4), environment-specific arguments (Table 5), replay buffer (Section B.5), and evaluation methodology (Section B.7), please refer to the corresponding descriptions in the DQN baseline implementation details (Section B).

Key distinctions of the TempoRL agent include its specialized network architecture with separate heads for action selection and skip duration, unique hyperparameters related to these mechanisms (`skip_dim`, `weight_sharing`), and the use of Huber loss for training.

D.1 Model Architecture

The TempoRL network processes input observations to produce Q-values for primitive actions and Q-values for skip durations. The input observation from the maze environments has a shape (C, H, W), where $C = 3$, identical to the DQN baseline (Section B.1).

D.1.1 Convolutional Neural Network (CNN) Part

The CNN component used for initial feature extraction is identical to the one in the DQN baseline. For details on its architecture (number of layers, channels, kernel sizes, strides, and activations), please refer to Section B.1. The output of this CNN part is a flattened 1D vector of size $n\_flatten$.

D.1.2 Linear Heads for Action and Skip Policies

Following the CNN, the network processes features through a structure that leads to two distinct output heads: one for action selection and one for determining the skip duration. The MLP for each pathway (from CNN output to pre-output layer) consists of layers with output units [512, 128, 32].

- **Feature Processing and Weight Sharing**:
  - If `weight_sharing = True` (default configuration): The $n\_flatten$ vector is first processed by a shared linear layer producing 512 output units, followed by a Leaky ReLU activation (negative slope 0.1). This 512-unit feature vector serves as the common input to the subsequent differing layers of the action and skip heads.
  - If `weight_sharing = False`: The $n\_flatten$ vector is independently fed into the first linear layer (512 output units, Leaky ReLU) of both the action and skip processing streams. Each stream then continues with its own [128, 32] layers.
- **Action Head**:
  - Starting from the 512-unit feature vector (either shared or head-specific), it is processed through two subsequent linear layers with 128 and 32 output units, respectively. Each of these hidden layers uses a Leaky ReLU activation (negative slope 0.1).
  - The final linear layer of the action head maps the 32-unit feature vector to $n\_output\_actions$ Q-values, where $n\_output\_actions = 4$ (corresponding to right, left, up, down).
- **Skip Head**:
  - Similarly starting from the 512-unit feature vector, it is processed through two subsequent linear layers with 128 and 32 output units, each followed by a Leaky ReLU activation (negative slope 0.1).
  - The final linear layer of the skip head maps the 32-unit feature vector to `skip_dim`. Each corresponds to the utility of repeating the chosen primitive action for a specific number of steps, from 1 up to `skip_dim`.

D.2 Hyperparameters

General experimental configuration (Table 4) and environment-specific arguments (Table 5) are consistent with the DQN baseline. Algorithm-specific hyperparameters for TempoRL, including those unique to its architecture, are detailed in Table 8.

D.3 Action Selection and Exploration

TempoRL employs a two-step ϵ-greedy strategy for exploration and action selection:

1. **Primitive Action Selection**: Given the current state s_t, a primitive action a_t (e.g., right, left, up, down) is chosen. With probability ϵ, a_t is selected randomly from the set of $n\_output\_actions$. Otherwise (with probability $1-\epsilon$), $a_t = \text{argmax}_{a'} Q(s_t, a'; \theta)$, where $Q(s_t, \cdot; \theta)$ are the Q-values produced by the action head of the online network.

2. **Skip Duration Selection**: Conditioned on the current state s_t and the chosen primitive action a_t, a skip duration k_t (number of times to repeat a_t, from 1 to `skip_dim`) is selected. With probability ϵ, k_t is chosen randomly from $\{1, \ldots, \texttt{skip\_dim}\}$. Otherwise, $k_t = \text{argmax}_{k'} Q_{skip}(s_t, a_t, k'; \theta_{skip})$, where $Q_{skip}(s_t, a_t, \cdot; \theta_{skip})$ are the Q-values for different skip durations produced by the skip head (which might use shared parameters if `weight_sharing = True`).

The selected primitive action a_t is then executed in the environment for k_t consecutive timesteps. The value of ϵ is typically linearly annealed from `start_e` to `end_e` over `exploration_fraction` of total timesteps, as detailed for the DQN baseline (see Section B.3 and Table 8).

Table 8: Algorithm Specific Hyperparameters for TempoRL Baseline

Parameter	**Default Value / Options**
total_timesteps	Environment specific
learning_rate	$[1 \times 10^{-3}, 1 \times 10^{-4}]$
buffer_size	[10000, 50000]
gamma (γ)	0.99 (discount factor)
tau (τ)	[0.01, 0.005] (target network update rate)
target_network_frequency	[10, 100] (frequency of applying soft target network update)
batch_size	256
start_e	1.0 (starting epsilon for exploration)
end_e	0.1 (ending epsilon for exploration)
exploration_fraction	[0.1, 0.3, 0.5]
learning_starts	1000 (timestep to start learning)
train_frequency	2 (frequency of training the Q-network)
activation_function	Leaky ReLU (negative slope 0.1 for hidden layers)
skip_dim	10 (maximum skip size)
weight_sharing	True (whether to share the first 512-unit layer)

D.4 Optimizer

Separate Adam optimizers (`torch.optim.Adam`) are used for the action Q-network parameters and the skip Q-network parameters. The learning rate for both optimizers is controlled by the `learning_rate` hyperparameter (see Table 8). Gradients for both networks are clipped (`grad_clip_val = 40.0`).

D.5 Replay Buffers

TempoRL utilizes two distinct replay buffers with capacity `buffer_size` (see Table 8) to store experiences for training its action and skip policies:

- **Action Replay Buffer**: This is a standard replay buffer (`ReplayBuffer` from Stable Baselines3) that stores transitions corresponding to individual primitive actions. Each experience tuple is of the form $(s_t, a_t, r_t, s_{t+1}, d_t)$, where:
  - s_t: The state at time t.
  - a_t: The primitive action taken at time t.
  - r_t: The reward received at time $t+1$.
  - s_{t+1}: The state at time $t+1$.
  - d_t: A boolean flag indicating if s_{t+1} is a terminal state.

  Experiences sampled from this buffer are used to train the action Q-network (the action head).

- **Skip Replay Buffer**: This is a custom replay buffer (referred to as `NoneConcatSkipReplayBuffer` in the implementation) specifically designed to store experiences related to the execution of multi-step skip actions. Each experience tuple is of the form $(s_j, k_j, s_{j+k_j}, R_j, d_{j+k_j}, k_j^{len}, a_j^{behav})$, representing:
  - s_j: The state from which the skip action (repeating a_j^{behav}) commenced.
  - k_j: The selected skip duration (i.e., the 'action' taken by the skip policy).
  - s_{j+k_j}: The state reached after the primitive action a_j^{behav} was executed k_j^{len} times.
  - R_j: The accumulated (and potentially discounted, depending on exact calculation before storage) reward received over the course of the k_j^{len} steps of the skip.
  - d_{j+k_j}: A boolean flag indicating if s_{j+k_j} (the state after the skip) is a terminal state.
  - k_j^{len} (length): The actual number of steps the primitive action a_j^{behav} was repeated (this is equivalent to k_j).

  - a_j^{behav}: The underlying primitive action that was chosen to be repeated for k_j^{len} steps.

  Experiences sampled from this buffer are used to train the skip Q-network (the skip head).

D.6 Training Details

TempoRL involves separate training updates for the action Q-network and the skip Q-network, both utilizing the Huber loss function.

D.6.1 Loss Function and Updates

The network parameters are updated by minimizing the Huber loss (This is implemented via torch.nn.SmoothL1Loss) for both action and skip predictions.

- **Action Q-Network Update**: Experiences $(s_j, a_j, r_j, s_{j+1}, d_j)$ are sampled from a standard replay buffer. The target value y_j^{action} is computed using a Double DQN-style approach:

$$y_j^{action} = r_j + \gamma(1 - d_j)Q_{target}(s_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'; \theta); \theta^-)$$

  where Q is the online action Q-network with parameters θ, and Q_{target} is its target network with parameters θ^-. The loss is then:

$$L_{action}(\theta) = \mathbb{E}_{(s_j,a_j,r_j,s_{j+1},d_j)\sim\mathcal{B}}\left[\text{HuberLoss}(y_j^{action} - Q(s_j, a_j; \theta))\right]$$

- **Skip Q-Network Update**: Experiences $(s_j, a_j^{behav}, k_j, R_j, s_{j+k_j}, d_{j+k_j})$ are sampled from a separate replay buffer for skips. Here, a_j^{behav} is the primitive action executed, k_j is the skip duration (number of times a_j^{behav} was repeated), R_j is the accumulated discounted reward during these k_j steps, and s_{j+k_j} is the state after k_j steps. The target value y_j^{skip} is calculated as:

$$y_j^{skip} = R_j + \gamma^{k_j}(1 - d_{j+k_j})Q_{target}(s_{j+k_j}, \arg\max_{a'} Q(s_{j+k_j}, a'; \theta); \theta^-)$$

  Note that the future value component $Q_{target}(s_{j+k_j}, \dots)$ uses the main action Q-network and its target, reflecting the value of the optimal next primitive action after the skip concludes. The current prediction is $Q_{skip}(s_j, a_j^{behav}, k_j; \theta_{skip})$, where Q_{skip} has parameters θ_{skip} (which may share some parameters with θ if `weight_sharing = True`). The loss is:

$$L_{skip}(\theta_{skip}) = \mathbb{E}_{(\dots)\sim\mathcal{B}_{skip}}\left[\text{HuberLoss}(y_j^{skip} - Q_{skip}(s_j, a_j^{behav}, k_j; \theta_{skip}))\right]$$

Both loss functions are optimized using their respective Adam optimizers, and gradients are clipped to prevent large updates.

D.6.2 Target Network Updates

To stabilize training, target networks are employed.

- A target network Q_{target} (with parameters θ^-) is maintained for the primary action Q-network Q (parameters θ). This Q_{target} is always updated using Polyak averaging: $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$.
- If `weight_sharing` is 'False', the distinct skip Q-network Q_{skip} (parameters θ_{skip}) has its own separate target network, Q_{skip_target} (parameters θ_{skip}^-). This Q_{skip_target} is similarly updated using Polyak averaging with θ_{skip} and θ_{skip}^-.
- If `weight_sharing` is 'True', the parameters of the skip mechanism are part of the overall network structure whose online parameters are θ (which includes the shared trunk and potentially specific skip head layers not part of the action head). In this scenario, a separate Polyak update for a distinct Q_{skip_target} is not performed; target values for the skip component's loss are derived using Q_{target} for estimating future state-action values, as shown in the skip target formula.

The soft update rate τ and the update frequency `target_network_frequency` are specified in Table 8.

E Proto Sequence Decoder (PSD) Implementation Details

This section outlines the architecture and configuration of the Proto-Sequence Decoder (PSD) module used in our model. It describes the decoder network structure, training objectives, latent space regularization, sequence reconstruction process, and other relevant design choices that enable effective decoding of proto-sequence embeddings into action-sequence.

E.1 Training Dataset Setup

To ensure that the model learns from meaningful and structured data rather than arbitrary noise, we constructed the training dataset according to the following constraints:

1. **Sequence Length Constraint**: All action sequences have lengths between 1 and $L_{max} = 10$ steps.
2. **Action Diversity Constraint**: Each sequence includes at most **two distinct action types**. For example, valid sequences include $[\texttt{up}, \texttt{up}]$ or $[\texttt{up}, \texttt{left}]$, whereas a sequence like $[\texttt{up}, \texttt{left}, \texttt{down}]$ is considered invalid.
3. **Switch Constraint**: Each sequence may contain at most **one switch between action types**. For instance, $[\texttt{up}, \texttt{left}]$ is allowed, but $[\texttt{up}, \texttt{left}, \texttt{up}]$ is not.
4. **Directional Conflict Constraint**: Sequences cannot include **opposite directions**, such as both `up` and `down`, or both `left` and `right`.
5. **Avoidance of loops**.

Following these criteria, we generated a total of **400 unique action sequences**. These sequences were one-hot encoded and padded with EOS tokens to a fixed length of L_{max} to suit VAE input requirements. These sequences form the basis of the training data for the Proto-Sequence Decoder (PSD).

E.2 Model Architecture

The Proto-Sequence Decoder (PSD) is implemented as a Variational Autoencoder (VAE) designed to map proto-sequence embeddings into action sequences of varying length. It is trained using reconstruction loss combined with a Kullback-Leibler divergence (KLD) regularization toward a standard Gaussian prior. The decoder operates on flattened one-hot sequence representations of actions and outputs reconstructed sequences over a predefined action vocabulary.

E.2.1 Input Representation

Each action in the sequence is represented as a one-hot vector over a vocabulary of size $n_{\text{words}} = 5$ corresponding to `{up, down, right, left, eos_token}`. The decoder models sequences of up to `input_length` $= 10$ actions, resulting in an input vector of dimension $10 \times 5 = 50$.

E.2.2 Encoder Network

The encoder receives a flattened 50-dimensional input vector and passes it through a series of fully connected layers:

1. **Linear Layer 1**:
   - Input size: 50
   - Output size: 32
   - Normalization: InstanceNorm1d
   - Activation: LeakyReLU (slope 0.2)

2. **Linear Layer 2**:
   - Output size: 16
   - Normalization: InstanceNorm1d
   - Activation: LeakyReLU
3. **Linear Layer 3**:
   - Output size: 16
   - Normalization: InstanceNorm1d
   - Activation: Tanh

The output is then projected into two parallel linear layers to produce the latent mean $\mu \in \mathbb{R}^{16}$ and log-variance $\log \sigma^2 \in \mathbb{R}^{16}$. A latent sample z is drawn using the reparameterization trick: $z = \mu + \sigma \cdot \epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$.

E.2.3 Decoder Network

The sampled latent vector $z \in \mathbb{R}^{16}$ is decoded through a symmetric feedforward network:

1. **Linear Layer 1**:
   - Output size: 16
   - Normalization: InstanceNorm1d
   - Activation: LeakyReLU
2. **Linear Layer 2**:
   - Output size: 32
   - Normalization: InstanceNorm1d
   - Activation: LeakyReLU
3. **Linear Layer 3**:
   - Output size: 50 (reconstructed sequence)
   - Normalization: InstanceNorm1d
   - Final Activation: Sigmoid (applied element-wise)

E.3 Training Objective

The PSD is optimized using a combination of:

- **Reconstruction Loss**: Binary cross-entropy loss between the input sequence and its reconstruction, normalized by sequence length.
- **KL Divergence Loss**: Encourages the latent distribution to match a unit Gaussian prior.

All input sequences are EOS-padded to the maximum length of 10 to ensure uniform input dimensionality across batches.

E.4 Hyperparameters

The training of the Proto-Sequence Decoder (PSD) is governed by a set of fixed hyperparameters, detailed in Table 9. These parameters control aspects such as optimization, batch processing, and reproducibility.

Table 9: Proto-Sequence Decoder (PSD) Training Hyperparameters

Parameter	**Value**
`train_on_entire_dataset`	True
`seed`	42
`optimizer`	Adam
`optimizer_learning_rate`	1×10^{-4}
`optimizer_weight_decay`	1×10^{-3}
`batch_size`	32

E.5 Optimizer

The PSD is trained using the Adam optimizer (`torch.optim.Adam`). The learning rate is controlled by the `learning_rate` hyperparameter (see Table 9).

E.6 Training Details

E.6.1 Loss Function

The Proto-Sequence Decoder (PSD) parameters are updated by minimizing a combined loss:

$$L = L_{\text{rec}} + L_{\text{KL}},$$

where:

- L_{rec} is the label-smoothed binary cross-entropy over the reconstructed sequence:

$$L_{\text{rec}} = -\frac{1}{T}\sum_{t=1}^{T}\Big[y_t \log \hat{y}_t + (1 - y_t)\log(1 - \hat{y}_t)\Big],$$

  with y_t replaced by $\tilde{y}_t = y_t(1-\epsilon) + \frac{\epsilon}{2}$, $\epsilon = 0.1$, and $T = 10$ is the sequence length.

- L_{KL} is the Kullback–Leibler divergence between the approximate posterior and a unit Gaussian:

$$L_{\text{KL}} = D_{\text{KL}}\big(\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0, I)\big).$$

E.6.2 Training Procedure

- **Maximum Steps:** Train for up to 20,000 epochs.
- **Batch Composition:** Split sequences into
  - *Short* (≤ 5 actions) and
  - *Long* (> 5 actions),

  and sample each batch with a 50/50 ratio of short and long sequences.

E.7 Evaluation

The Proto-Sequence Decoder's (PSD) performance was assessed on the entire training set using two key metrics. Evaluations were conducted every 50 epochs, and the checkpoint yielding the highest exact match accuracy was retained. After 20,000 epochs, the following results were achieved:

- **Exact Match Accuracy:** This metric measures the proportion of sequences reconstructed with zero errors. The PSD achieved an Exact Match Accuracy of 0.978.
- **Per-Step Accuracy:** This metric calculates the fraction of correctly reconstructed actions across all positions within the sequences. The PSD achieved a Per-Step Accuracy of 0.99.

E.8 Visualization of the Learned Embedding Space

Figure 5 illustrates the two-dimensional t-SNE projection of the learned proto-action-sequence embeddings. Each point corresponds to one sequence from the dataset, with colors representing the effective sequence length. As can be seen, sequences with similar structural properties tend to form dense clusters, indicating that the embedding space preserves meaningful relationships between sequences.

A particularly noteworthy observation is the position of the red star, which represents a previously unseen sequence not included during training. This sequence is located within the cluster of its closest structural neighbors, suggesting that the learned representation generalizes effectively to new data. In other words, the embedding model is able to position novel sequences near the most similar examples from the training set, supporting its potential for robust retrieval, similarity search, and downstream predictive tasks.

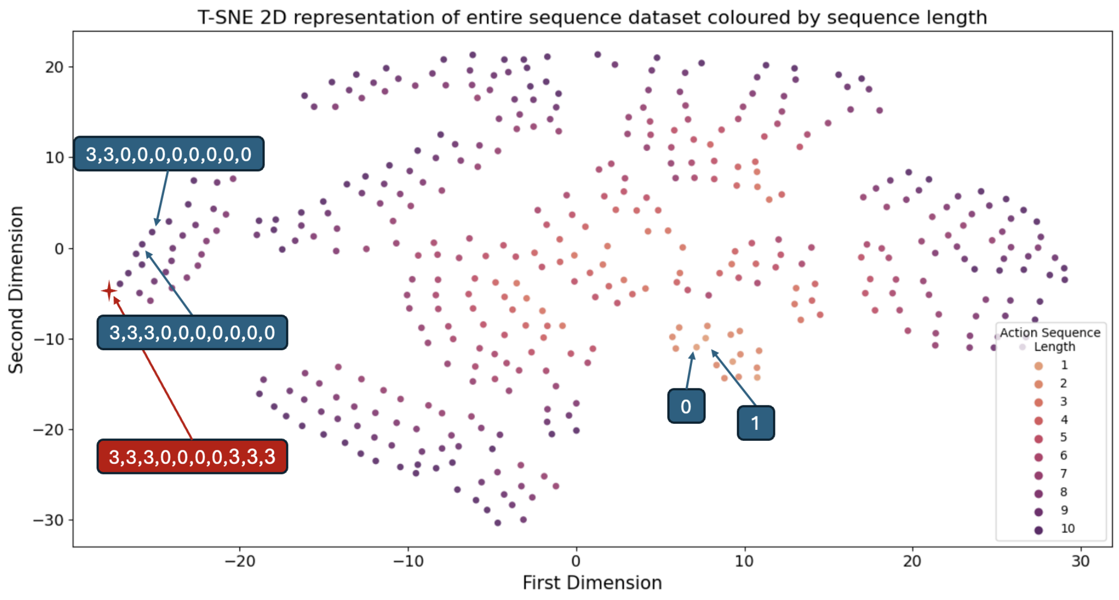

Figure 5: T-SNE 2-D projection of the proto-action-sequence embedding space. The map contains embeddings for the **400 original sequences** in the dataset together with **one previously unseen sequence** (red star). Points are colored by their effective sequence length, and sequences with similar structure form tight neighborhoods. The unseen sequence falls naturally inside the cluster of its closest structural neighbors, showing that the learned representation places **new, out-of-training sequences adjacent to the most similar known examples.**

F GPS (Generative Proto Sequence) Implementation Details

This section outlines the architecture and configuration of our GPS (Generative Proto Sequence) method. GPS is an actor-critic based algorithm where the actor network generates a latent representation, termed "proto-sequence." This proto-sequence is then processed by a pre-trained generative decoder model (PSD) to produce a sequence of discrete actions. A critic network evaluates this action sequence to guide learning. The subsequent subsections detail neural network structures, hyperparameter settings, exploration strategy, optimization methods, and other relevant training aspects.

F.1 Model Architecture and State Representation

The GPS agent consists of three primary neural network components: an Actor, a Critic, and a pre-trained Decoder. It processes observations from the environment. For details on the specific state representation, input shape (e.g., height, width, channels), and preprocessing, please refer to the "State Representation" within Section 4.3.

F.1.1 Convolutional Neural Network (CNN) Part

Both the Actor and Critic networks use separate but identical Convolutional Neural Network (CNN) architecture to extract features from the input observation. The CNN component used for initial feature extraction is identical to the one in the DQN baseline. For details on its architecture (number of layers, channels, kernel sizes, strides, and activations), please refer to Appendix B.1. The output of this CNN part is a flattened 1D vector of size $n\_flatten$.

Note: in our current implementation, the actor and critic networks each have their own separate CNN state feature extractors. In future work, we plan to explore a shared CNN feature extraction architecture as implemented in TempoRL, which could potentially improve computational efficiency and state representation learning.

F.1.2 Actor Network

The Actor network takes the extracted features from the CNN and produces a proto-sequence embedding.

- **Input**: The flattened feature vector n_{features} from the CNN is concatenated with a positional encoding. The embedding dimension of this positional encoding is specified by `pe_embedding_dim`.
- **Architecture**: The combined features are processed through a series of fully connected linear layers, defined by the `actor_linear_layers` parameter (e.g., [512, 128, 32]). The activation function for these hidden layers is specified by `actor_linear_layers_activation_function` which is "leaky_relu" with negative slope 0.1.
- **Output**: The actor generates a single proto-sequence embedding. This embedding is a vector of size `actor_n_output_channels` and serves as input to the Decoder PSD network.

F.1.3 Position Encoding in Proto-Sequence Generation

To enhance the expressiveness of our action sequence generation, we incorporate positional encoding within the Actor network. This technique, inspired by transformer architectures (Dosovitskiy et al., 2020), helps the Actor generate more contextually aware proto-sequence embeddings by providing explicit spatial information about the agent and goal positions.

Implementation Details. Our positional encoding implementation combines both row and column information for each grid cell in the observation space:

1. We create sinusoidal encodings for both dimensions (height and width) separately:

   ```
   pe_row[:, 0::2] = torch.sin(position_row * div_term)
   pe_row[:, 1::2] = torch.cos(position_row * div_term)
   ```

2. These encodings are combined into a unified representation where the first half of each cell's embedding encodes its row position and the second half encodes its column position.

3. During forward passes, we extract the agent and goal positions from the observation and retrieve their respective positional encodings:

   ```
   agent_pe = self.position_encoding[agent_row, agent_col]
   goal_pe = self.position_encoding[goal_row, goal_col]
   ```

4. These position-specific features are concatenated with the CNN-extracted features before being processed by the linear layers of the Actor.

Motivation and Benefits. Integrating positional encoding within the Actor network provides several advantages:

1. **Enhanced Spatial Reasoning**: By explicitly encoding agent and goal positions, the Actor can better understand spatial relationships, which is crucial for navigation tasks.
2. **Improved Exploration Early in Training**: The position encodings enable the generation of more diverse proto-sequence embeddings in the initial training phases, facilitating better exploration before the CNN features become well-trained.
3. **Direction-Aware Sequence Generation**: The relative positions of agent and goal inform the Actor about the general direction of movement required, allowing it to generate more purposeful action sequences even with limited experience.
4. **Invariance to Visual Feature Quality**: Especially early in training when the CNN features may be unreliable, position encodings provide a stable signal that enables meaningful action sequence generation.

Our manual investigations and targeted experiments suggest that incorporating position encodings enhances the model's capabilities in several ways. We observed that the position-enriched Actor generates proto-sequence embeddings with greater contextual awareness of spatial relationships, which in turn produces more diverse and situation-appropriate action sequences. Without position encoding, the proto-sequence embeddings tended to cluster more closely in the latent space, resulting in less differentiated action patterns. This difference was particularly evident in larger and more complex maze environments, where the position-encoded model demonstrated an improved ability to generate directionally coherent sequences that efficiently navigated toward goals. The positional information appears to provide a structural prior that helps the Actor formulate meaningful navigation strategies even before the CNN features are fully refined through training.

F.1.4 Critic Network

The Critic network estimates the Q-value of a state and a decoded action sequence.

- **Input**:
  - The flattened feature vector n_{features} from the CNN, representing the current state.
  - The action sequence generated by the Decoder from the actor's proto-sequence. The representation used for this action sequence is `ACTION_SEQ_AS_ONE_HOT`. Namely, the input is a tensor where each action in the sequence of length `n_actions_in_seq` is one-hot encoded. Shorter sequences are padded to this length using an End-of-Sequence (EOS) token. Each one-hot vector has a dimension equal to the `action_space_size` plus one (for the EOS token). Consequently, the total input dimension for the action sequence part, `action_seq_dim`, is `n_actions_in_seq` $\times$ (`action_space_size` + 1).
- **Architecture**: The inputs are processed through a series of fully connected linear layers, defined by `critic_linear_layers` (e.g., [512, 128, 32]). The activation function is specified by `critic_linear_layers_activation_function` which is "leaky_relu" with negative slope 0.1.
- **Output**: A single scalar Q-value. The output Q-value can be optionally clipped between `min_qf_value` and `max_qf_value`.

F.1.5 Decoder Network

A pre-trained generative model, specifically a Variational Autoencoder, acts as the Decoder (PSD).

- **Loading**: The Decoder is loaded from a pre-trained model specified by `decoder_model_path`.
- **Input**: The proto-sequence embedding (size `actor_n_output_channels`) generated by the Actor.
- **Output**: A sequence of `n_actions_in_seq` discrete actions. Each action is selected from a vocabulary of `action_space_size` primitive actions plus an `end_of_sequence_token` token.

- **Generation**: The Decoder can use Gumbel-Softmax for differentiable sampling if `use_gumble_in_decoder` is true, or a deterministic argmax with a Straight-Through Estimator otherwise. See explanation in Section 4.2.

F.2 Hyperparameters

The GPS method is configured by a wide range of hyperparameters. General experiment settings and environment-specific configurations are typically managed as detailed for the DQN baseline (see Tables 4 and 5). Key algorithm-specific hyperparameters for GPS are listed in Tables 10, 11.

Table 10: Algorithm Specific Hyperparameters for GPS

Parameter	**Value / Options / Description**
`total_timesteps`	Env. specific
`actor_learning_rate`	Actor LR (e.g., 1×10^{-4}).
`critic_learning_rate`	Critic LR (e.g., 1×10^{-4}).
`buffer_size`	[10000, 50000]
`gamma` (γ)	Discount factor.
`tau` (τ)	Target net. soft update rate (e.g., 0.005).
`batch_size`	Experiences per train step.
`learning_starts`	Timestep train begins.
`actor_policy_frequency`	Actor net. update freq. rel. to Critic (e.g., 2).
`actor_target_network_frequency`	Target Actor net. update freq. (e.g., 10 steps).
`critic_target_network_frequency`	Target Critic net. update freq. (e.g., 10 steps).
`start_e`	Initial ϵ for ϵ-greedy (e.g., 1.0).
`end_e`	Final ϵ value (e.g., 0.1).
`total_steps_e`	Timesteps for ϵ annealing (e.g., 15000).
`sub_sequences_move_start_point`	Boolean; varying sub-seq. start points.
`sub_sequences_move_end_point`	Boolean; varying sub-seq. end points.
`sub_sequences_min_jump_move_start_point`	Min. jump for start-moved sub-seq. gen.
`sub_sequences_min_jump_move_end_point`	Min. jump for end-moved sub-seq. gen.
`every_one_step_transition_to_buffer`	Boolean; all single-step trans. stored.
`actor_n_output_channels`	Proto-action-seq. embed dim. (e.g., 16).
`actor_linear_layers`	Actor MLP hidden layer sizes (e.g., [512, 128, 32]).
`actor_linear_layers_activation_function`	Actor MLP activation (e.g., "leaky_relu").
`actor_weight_decay`	Actor L2 reg. strength (e.g., 1×10^{-2}).
`pe_embedding_dim`	Positional encoding dim. in Actor (e.g., 16).
`critic_linear_layers`	Critic MLP hidden layer sizes (e.g., [512, 128, 32]).
`critic_linear_layers_activation_function`	Critic MLP activation (e.g., "leaky_relu").
`critic_weight_decay`	Critic L2 reg. strength (e.g., 1×10^{-3}).
`decoder_model_path`	Path to pre-trained VAE Decoder model.
`n_actions_in_seq`	Decoder action seq. length (e.g., 10).
`action_space_size`	Unique primitive actions in env. (e.g., 4).
`end_of_sequence_token`	Decoder integer token for end of seq. (e.g., 4).
`use_gumble_in_decoder`	Boolean; Decoder uses Gumbel-Softmax.

F.3 Reward Function and Empty Sequence Handling

The GPS agent utilizes a structured reward function designed to encourage efficient navigation while penalizing inefficient or invalid behaviors. The reward function is defined as:

$$R = r_{\text{goal}} - \frac{1}{l_{\max}} \times n_{\text{valid}} - \frac{3}{l_{\max}} \times n_{\text{invalid}}$$

Table 11: GPS Hyperparameters Settings

Parameter	**Value**
`total_timesteps`	Environment specific
`buffer_size`	`[10000, 50000]`
`gamma`	0.99
`tau`	`[0.01, 0.005]`
`batch_size`	256
`start_e`	1
`end_e`	0.1
`learning_starts`	1000
`actor_learning_rate`	`[1e-3, 1e-4, 1e-5]`
`critic_learning_rate`	`1e-04`
`actor_policy_frequency`	2
`sub_sequences_move_start_point`	`TRUE`
`sub_sequences_move_end_point`	`TRUE`
`sub_sequences_min_jump_move_start_point`	1
`sub_sequences_min_jump_move_end_point`	1
`every_one_step_transition_to_buffer`	`TRUE`
`actor_target_network_frequency`	`[10, 100]`
`critic_target_network_frequency`	10
`total_steps_e`	15000
`actor_n_output_channels`	16
`actor_linear_layers`	`["512, 32", "512, 128, 32"]`
`actor_linear_layers_activation_function`	`leaky_relu` (negative slope 0.1)
`actor_weight_decay`	`1e-04`
`pe_embedding_dim`	128
`critic_linear_layers`	`[512, 128, 32]`
`critic_linear_layers_activation_function`	`leaky_relu` (negative slope 0.1)
`critic_weight_decay`	`1e-04`
`max_level`	Environment specific
`start_level`	Environment specific
`use_gumble_in_decoder`	`TRUE`

where r_{goal} is 1 if the agent reached the goal (0 otherwise), $l_{\max}$ is the maximal start-goal distance acting as a regularizer, n_{valid} is the number of valid actions taken, and n_{invalid} is the number of invalid actions (e.g., bumping into a wall).

To handle cases during training where the actor generates proto-sequences that are decoded as empty sequences (i.e., where the PSD outputs the EOS token as the first action), we apply a harsh penalty of -20. For these instances, we also hard-code the action to be 1 (DOWN) to ensure the agent always takes some action. This direct negative reinforcement was found to be highly effective in guiding the actor to produce valid proto-sequence embeddings that decode into meaningful action sequences. Without this penalty, the actor might frequently produce embeddings that map to empty, significantly hampering exploration and learning progress. This approach provides a clear signal to the actor network about the importance of generating proto-sequences that translate to substantive action sequences, accelerating the learning process and improving the overall stability of training. Empirically, we observed that this simple yet effective mechanism substantially reduced the occurrence of empty sequences.

F.4 Action Selection and Exploration

At each decision step, the Actor network generates a proto-sequence embedding. The Decoder then translates this embedding into a corresponding action-sequence of primitive actions. The agent employs an ϵ-greedy exploration strategy:

- With probability ϵ_t (where ϵ_t anneals from `start_e` to `end_e` over `total_steps_e` steps): An exploratory action sequence is selected. This sequence is typically chosen randomly from a pre-defined set of valid action sequences see Section 4.3 for more details.
- With probability $1 - \epsilon_t$ (exploitation): The action sequence generated by the Actor-Decoder pipeline is used for execution.

The value of ϵ_t is linearly annealed:

$$\epsilon_t = \max\Big(end\_e, start\_e - (start\_e - end\_e) \cdot (\text{current\_step}/total\_steps\_e)\Big)$$

The chosen action sequence is subsequently trimmed using the `end_of_sequence_token` before execution.

F.5 Optimizer

The Actor and Critic networks are trained using separate Adam optimizers (`torch.optim.Adam`).

- The Actor's optimizer is configured with a learning rate of `actor_learning_rate` and applies L2 weight decay with a coefficient of `actor_weight_decay`.
- The Critic's optimizer uses a learning rate of `critic_learning_rate` and L2 weight decay with a coefficient of `critic_weight_decay`.

F.6 Replay Buffer

A replay buffer (`ReplayMemory`) with a capacity of `buffer_size` stores past experiences. Each stored transition typically includes: the current state observation (s_t), the next state observation (s_{t+1}), the selected action sequence (act_seq_t), the received reward (r_t), a terminal flag (d_t), and the actor's proto-action-sequence embedding that generated act_seq_t (emb_t). The system may also store sub-sequences derived from executed plans if parameters such as `push_every_one_step_transition_to_buffer`, `push_sub_sequences_to_buffer_move_start_point`, and `push_sub_sequences_to_buffer_move_end_point` are enabled, potentially enriching the diversity of experiences in the buffer.

F.7 Training Details

Network training commences after `learning_starts` timesteps have been collected. Updates are performed using batches of `batch_size` experiences sampled from the replay buffer.

F.7.1 Critic Network Update

The Critic network parameters (θ_C) are updated by minimizing the Mean Squared Error (MSE) loss:

$$L(\theta_C) = \mathbb{E}_{(s,\text{act\_seq},r,s',d,\text{emb})\sim\mathcal{B}}\left[(Q(s, \text{emb}, \text{act\_seq}; \theta_C) - y_t)^2\right]$$

The target Q-value y_t is computed using the target Actor (`target_actor_network`) and target Critic (`target_critic_network`) networks to ensure stability:

$$y_t = r + \gamma(1-d)Q_{\text{target}}\big(s', \text{Actor}_{\text{target}}(s'), \text{Decoder}(\text{Actor}_{\text{target}}(s')); \theta_C^-\big)$$

where $\text{Actor}_{\text{target}}(s')$ is the proto-sequence embedding from the target actor for state s', $\text{Decoder}(\cdot)$ converts it to an action sequence, and θ_C^- are the parameters of the target critic. The Q-values from the target critic can be clipped using `min_qf_value` and `max_qf_value`.

F.7.2 Actor Network Update

The Actor network parameters (θ_A) are updated with a frequency of `actor_policy_frequency` (delayed policy update). The goal is to adjust the actor's parameters to produce a proto-sequence that leads to a higher Q-value as estimated by the current Critic. For a sampled batch of states, the Actor generates a proto-sequence embedding. This is decoded into an action sequence, which is then evaluated by the online Critic network $Q(\cdot; \theta_C)$. The actor loss is designed to maximize this Q-value:

$$L(\theta_A) = -\mathbb{E}_{s \sim \mathcal{B}} \left[Q(s, \text{Actor}(s), \text{Decoder}(\text{Actor}(s)); \theta_C) \right]$$

F.7.3 Target Network Updates

Separate target networks are maintained for both the Actor ($\text{Actor}_{\text{target}}$ with parameters θ_A^-) and the Critic (Q_{target} with parameters θ_C^-). Their parameters are updated using Polyak averaging with the parameters of their corresponding online networks (θ_A, θ_C):

$$\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$$

The soft update rate is τ. Target network updates for the Actor occur every `actor_target_network_frequency` steps, and for the Critic every `critic_target_network_frequency` steps.

F.7.4 Training Procedure Summary

- **Initialization**: Networks and target networks are initialized. Replay buffer is empty.
- **Data Collection**: Agent interacts with the environment using the action selection strategy (Section F.4), storing experiences ($s_t, s_{t+1}, \text{act\_seq}_t, r_t, d_t, \text{emb}_t$) in the replay buffer.
- **Learning Phase** (after `learning_starts` steps):
  1. Sample a `batch_size` of experiences from the replay buffer.
  2. Update Critic network parameters by minimizing the MSE loss with the computed TD targets.
  3. Periodically (every `actor_policy_frequency` steps), update Actor network parameters to maximize the Q-value of the generated sequence as estimated by the Critic.
  4. Periodically (every `actor_target_network_frequency` and `critic_target_network_frequency` steps respectively), update target Actor and target Critic networks using Polyak averaging.

F.8 Evaluation

The performance of the GPS agent is assessed periodically during training and/or at the end of the training process.

- **Frequency**: Evaluations on validation datasets typically occur every `val_eval_freq` steps, and on subsets of the training data every `train_eval_freq` steps. Less frequent evaluations may occur on a dedicated test dataset (e.g., every `eval_test_dataset_training_freq` steps) or at the end of training.
- **Method**: During evaluation, the actor generates a proto-sequence, the decoder converts it to an action sequence, and this sequence is executed. The Decoder may operate in a deterministic mode (`deterministic_inference = True`).
- **Metrics**: Standard reinforcement learning metrics are logged, such as mean episodic return and success rate. Additional metrics might include the average number of decoder generations per episode or properties of the generated action sequences.
- **Model Saving**: If `save_model` is enabled, the best performing models (actor and critic) are saved based on criteria defined by `save_model_strategy` (e.g., best success rate or mean reward on the validation set).

G Maze Evaluation Environments Benchmark

In our research, our evaluation environments consist of procedurally generated mazes with varying structures and complexity. We utilized synthetic maze environments created using Large Language Models (LLMs) to ensure unbiased benchmark construction. The following details our approach to maze generation for the different environment types used in our experiments.

G.1 Synthetic Maze Generation Process

We generated our maze environments using LLM. For each maze type (rooms, obstacles, and corridors), we provided specific prompts instructing the LLM to generate Python code that would create the maze environments according to our requirements. Importantly, all maze generation was performed programmatically without manual intervention, ensuring reproducibility and eliminating human bias. We specifically used the OpenAI o3-mini model for code generation, which was instructed to create five different variants for each maze type. To avoid experimenter bias in seed selection, we also employed an LLM to generate code for choosing random seeds:

```
import hashlib
def get_consistent_seed():
"""
Selects a seed number from a list consistently across multiple runs.
Returns:
int: The selected seed number.
"""
seed_list = [42, 1234, 9999, 2024, 2025]

Create a hash of the function name to ensure consistency
hash_object = hashlib.sha256(b'get_consistent_seed')
hash_value = int(hash_object.hexdigest(), 16)

Use the hash value to select a seed from the list
seed_index = hash_value % len(seed_list)
return seed_list[seed_index]

Get the consistent seed
seed = get_consistent_seed()
print(f"Selected seed: {seed}")
```

The code generated by the LLM for each maze type is available in our code repository. This approach ensured that the maze generation process was fully automated and free from experimenter bias, providing a consistent and fair benchmark for evaluating our GPS algorithm against the baselines.

G.2 Maze Type Generation Prompts

For each maze type, we provided detailed prompts to the LLM to guide the generation process:

G.2.1 Obstacles Maze Prompt

Maze Generation with 15% Obstacles
Write a Python script that programmatically generates five distinct 16×16 mazes with randomly placed obstacles. The grid follows these rules:
Grid & Output Format:

- The maze is a 16×16 grid.

- Each cell is either **open space (0)** or a **wall/obstacle (1)**.
- The **outermost border** (first and last rows and columns) **must remain walls (1)**.
- **15% of the inner cells** (excluding the border) should be randomly chosen as **obstacles (1)**, while the remaining are **open spaces (0)**.
- The final maze should be output as a Python dictionary:

```
{
    'maze': [
        "1111111111111111",  # Top row (wall frame)
        "1..............1",  # Use 0 for open spaces;
                             # dots are placeholders here.
        ...
        "1111111111111111"   # Bottom row (wall frame)
    ]
}
```

Replace the dots with appropriate **0s (open spaces)** and **1s (walls/obstacles)**.
Maze Generation Method: Initialize the Maze:

- Create a 16×16 grid where every cell is an **open space (0)**.
- The **outer border (first and last rows/columns) must always remain walls (1)**.
- The **inner** 14×14 **area** (excluding the border) will contain **open spaces (0) and obstacles (1)**.

Randomly Place Obstacles:

- **15% of the inner** 14×14 **cells** should be converted into **obstacles (1)**.
- The placement of these obstacles should be **random**.
- Ensure that at least one path remains between any two open spaces for potential connectivity.

Generate Five Distinct Mazes:

- Use different random seeds to create five unique mazes.
- Ensure that each maze has **exactly 15% obstacles** inside the inner area.

G.2.2 Rooms Maze Prompt

You are a maze designer responsible for enhancing a predefined base maze structure for a navigation simulation. Your task is to decide where entrances should be located while ensuring the maze meets the following requirements:
Maze Design Requirements Base Maze Structure:

- The maze is predefined and consists of a 16×16 grid.
- The base structure must remain intact, but you will determine the placement of the entrances and ensure connectivity between all rooms and open spaces.

Entrance and Room Connectivity:

- The maze is divided into **four equal-sized quadrants (rooms)** separated by walls.

- Each room must have only **two entrances of size 1**. Use a seeded random choice for entrance placement.
- Entrances should be placed strategically to ensure the maze is **fully connected**, meaning an agent can navigate between any two open cells (0) using **up, down, left, or right** movements.
- Passageways between rooms must be **narrow** and preserve the integrity of the maze's challenge.

Obstacle Coverage:

- Obstacles (1) must make up **5% of the total grid** (≈ 13 cells).
- Don't consider the obstacles that are part of the maze's frame.
- Obstacles may be added or removed **within constraints** to maintain connectivity and alignment with the entrance placement.
- Don't place obstacles in nearby squares close to any entrance. Make sure that an obstacle doesn't block any entrance.
- Use a seeded random choice for obstacles placement.

Reproducibility:

- Use a **specific random seed** to ensure the design is reproducible.

Output Format:

- Generate code for creating the maze.
- Generate the maze as a Python dictionary with a key (e.g., 'maze') and represent each row as a binary string.
- Output 5 mazes using different seeds and make sure that obstacles don't block entrances.

Base Maze Layout: The base structure is as follows:

```
{
   'base_maze': [
        "1111111111111111",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1111111111111111",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1000000100000001",
        "1111111111111111"
     ]
}
```

Design Task: Modify the maze by:

- Placing **entrances** in the walls separating the quadrants.
- The logic should be based on seed for determining the entrances position and obstacles positions.
- Ensuring the maze is **fully connected**.
- Making any minor adjustments to obstacles (1) to meet connectivity and percentage requirements.
- Verify by code that each room must have only **two entrances of size 1**.
- Verify by code that each obstacle doesn't block any entrance.

Output the modified maze as a Python dictionary with the format below:

```
{
    'maze': [
        "updated_row_1",
        "updated_row_2",
        ...,
        "updated_row_16"
    ]
}
```

General Steps:

- Choose two entrances for each room by selecting a random square on each wall.
- Randomly select a room and place obstacles within it until the obstacle budget is reached.

G.2.3 Corridors Maze Prompt

Write a Python script that programmatically generates five distinct 16×16 mazes. In each maze, start with a grid completely filled with wall cells (represented by 1), then carve out corridors by selecting one or more vertical lines and one or more horizontal lines to convert wall cells to open cells (represented by 0). The corridors will be 1-cell-wide, and they must intersect so that every open cell is reachable from any other via up, down, left, and right moves. The outer border of the maze should always remain as walls.

Grid & Output Format:

- The maze is a 16×16 grid.
- Each cell is either **open (0)** or a **wall (1)**.
- The **outermost border** (first and last rows and columns) **must remain walls**.
- The final maze should be output as a Python dictionary:

```
{
    'maze': [
        "1111111111111111",  # Top row (wall frame)
        "1..............1",  # Use 0 for open spaces;
                             # dots are placeholders here.
        ...
        "1111111111111111"   # Bottom row (wall frame)
    ]
}
```

Replace the dots with the appropriate 0s and 1s as per the carved corridors.
Corridor Carving Method: Initialize the Maze:

- Create a 16×16 grid where every cell is a **wall (1)**, with the outer border fixed as walls.

Select Corridor Lines:

- **Vertical Corridors:** Choose **2 up to 4** vertical columns (not including the outer borders) that will serve as corridors.
- **Restriction:** Ensure that no consecutive vertical columns are selected—there must be at least one wall column between any two chosen corridor columns.
- **Horizontal Corridors:** Choose **2 up to 4** horizontal rows (again, not including the outer borders) that will serve as corridors.
- **Restriction:** Ensure that no consecutive horizontal rows are selected—there must be at least one wall row between any two chosen corridor rows.
- These lines will form a network of corridors that cross each other.

Carve the Corridors:

- For each selected vertical column, change all cells in that column (except the outer border) from 1 (wall) to 0 (open space).
- Similarly, for each selected horizontal row, change all cells in that row (except the outer border) from 1 to 0.
- The intersections of these corridors (where a selected vertical column crosses a selected horizontal row) will naturally be open, ensuring connectivity.

Ensure Full Connectivity:

- The chosen vertical and horizontal corridors should intersect, guaranteeing that every open cell (in the corridors) is reachable from any other open cell.
- Optionally, you can add additional corridor "branches" (by clearing cells adjacent to the main corridors) to create a more interesting maze layout, as long as all open cells remain interconnected.

Randomness:

- Generate five distinct mazes by using different random seeds and varying the selected vertical and horizontal corridor positions.

H Analysis of Reward Strategy Impact on ASR

This appendix details the comparison of two step penalty strategies, illustrated in Figure 6. The strategies are the default Max-Level-based penalty ($-1/l_{\max}$, where $l_{\max}$ is a normalization factor related to task depth, e.g., $l_{\max} \approx 30$) and an alternative Map-Size-based penalty ($-1/\text{maze size}$, e.g., $-1/256$ for a 16×16 maze). Their relative efficacy is measured by $\Delta\text{ASR} = \text{ASR}_{\text{Max-Level}} - \text{ASR}_{\text{Map-Size}}$, where positive values indicate superior performance for the Max-Level strategy.

Figure 6 reveals distinct performance patterns across the tested environments.

- In simple 16 × 16 mazes, the Max-Level strategy provides a significant initial learning speedup (ΔASR $\approx +0.8$ at 200k steps), although its final Average Success Rate (ASR) is matched by the Map-Size strategy after approximately 400k training steps.
- When 15% obstacles are introduced in the 16 × 16 maze, increasing its complexity, the Max-Level strategy maintains a consistent performance advantage throughout the training. ΔASR peaks at approximately +0.44 and remains positive (settling around +0.1).
- In larger 24 × 24 mazes, the Max-Level strategy's superiority becomes more pronounced. ΔASR dramatically increases after 500k steps, reaching and sustaining a value of approximately +0.9. This highlights the diminishing effectiveness of the Map-Size penalty (e.g., $-1/576$ for 24 × 24) as it becomes increasingly diluted in larger state spaces.

The consistently superior performance of the Max-Level strategy, particularly in more complex or larger environments, can be attributed to several factors. Firstly, it provides a more **impactful and relevant penalty signal**. The $l_{\max}$-normalized penalty (e.g., $\approx -1/30$) offers a substantially stronger and more consistent learning feedback compared to the Map-Size penalty, which diminishes significantly with increasing maze size. Secondly, $l_{\max}$ serves as a normalization factor that likely **correlates better with the intrinsic task difficulty** and typical solution length than the raw cell count of the maze, which does not inherently capture navigational complexity. Consequently, the Max-Level penalty structure appears to offer more **effective exploration guidance** and promotes **greater learning efficiency**.

In summary, normalizing step penalties by $l_{\max}$ (Max-Level-based strategy) leads to a more robust and effective reward scheme for the navigation tasks studied. This approach fosters more efficient learning and achieves higher success rates by aligning the penalty signal more accurately with the inherent challenges of the environment, proving especially advantageous as task complexity and scale increase.

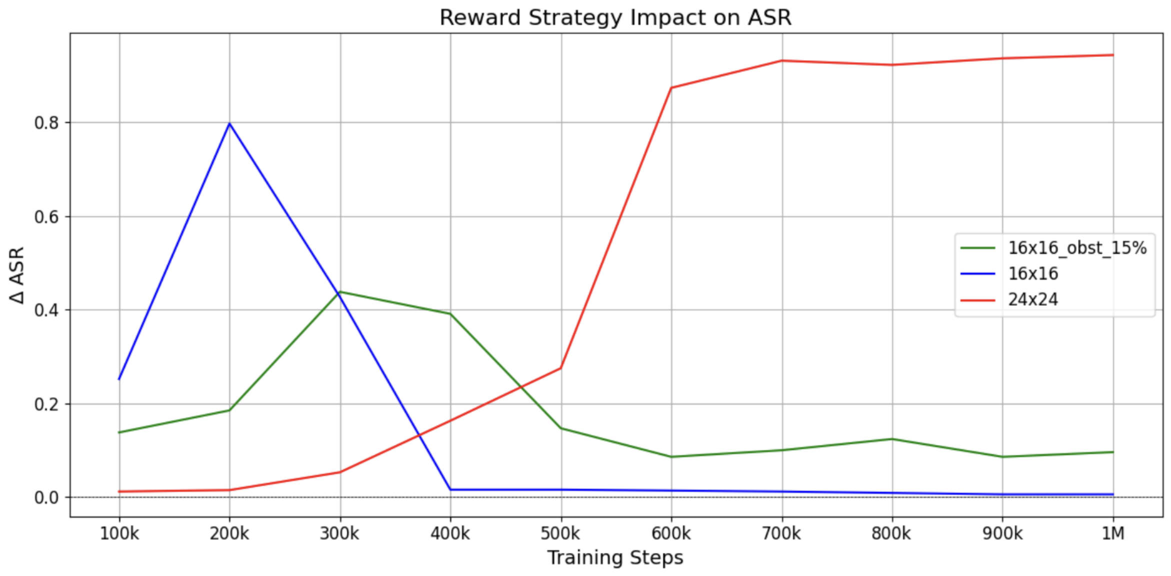

Figure 6: Comparison of average success rate differences (ΔASR) between two reward strategies: **Max-Level** (the default strategy) and **Map-Size**, which is identical except that it replaces $l_{\max}$ with the total number of cells in the maze (e.g., 256 for a 16×16 maze). The values are evaluated across training steps. A positive ΔASR indicates that the Max-Level reward strategy yields better performance.

I Evaluating the Average Success Rate (ASR) With Larger Train dataset

To assess performance on a larger dataset, we trained the agent on 2000 mazes. Table 12 presents the Average Success Rates (ASR) across various maze configurations.

Table 12: Average Success Rates (ASR) with 2000 mazes in train dataset across maze types.

Maze Type	**Max Steps**	**Train Size**	**Dist. from Start to Goal**	**DQN**	**GPS**	**GPS-D**	**TempoRL**	**DAR**
8x8	1M	2000	[1 – 14]	1.00	1.00	1.00	1.00	0.81
16_16	1M	2000	[16 – 26]	1.00	1.00	1.00	1.00	0.89
16x16_obst_15%	1.5M	2000	[20 – 30]	0.94	0.99	0.94	0.91	0.78
16x16_obst_25%	1.5M	2000	[20 – 30]	0.98	0.98	0.94	0.39	0.16
16x16_rooms	1.5M	2000	[20 – 30]	1.00	0.95	0.82	0.98	0.13
16x16_corr	1M	2000	[10 – 30]	1.00	1.00	1.00	1.00	0.67
24x24	1.5M	2000	[20 – 30]	0.98	1.00	0.98	1.00	0.75
24x24_obst_15%	1.5M	2000	[10 – 30]	0.05	0.09	0.05	0.03	0.13

J ASR Statistical Significance Testing

To assess the statistical significance of the differences in Average Success Rates (ASR) between our proposed method (GPS) and the baseline (DQN), we employed McNemar's test. This section details the methodology and presents the results of these tests.

Methodology

McNemar's test is a non-parametric test suitable for paired nominal data. It is used to determine whether there is a significant difference in the proportions of two related samples, such as when two algorithms are evaluated on the same set of test instances. In our context, each maze evaluation episode serves as a paired instance, and the outcome for each algorithm (GPS or DQN) is categorized as either a success or a failure.

An episode was deemed a **success** if the agent reached the goal in an episodic length of less than 75 steps. Otherwise, it was considered a **failure**.

For each pair of algorithms (GPS vs. DQN) on a given maze type, we constructed a 2×2 contingency table based on the outcomes of common evaluation episodes:

		Algorithm B (DQN)	
		Success	Failure
Algorithm A (GPS)	Success	a	b
	Failure	c	d

Where:

- a: Number of episodes where both GPS and DQN succeeded.
- b: Number of episodes where GPS succeeded and DQN failed.
- c: Number of episodes where GPS failed and DQN succeeded.
- d: Number of episodes where both GPS and DQN failed.

McNemar's test focuses on the discordant pairs (b and c). The null hypothesis (H_0) is that the two algorithms have the same ASR. The test statistic is calculated as:

$$\chi^2 = \frac{(b-c)^2}{b+c}$$

This statistic follows a chi-squared distribution with 1 degree of freedom. We used the version of the test without continuity correction, as implemented in 'statsmodels.stats.contingency_tables.mcnemar'.

The significance level was set at $\alpha = 0.05$. If the calculated p-value was less than 0.05, we rejected the null hypothesis and concluded that there is a statistically significant difference in the ASR performance of the two algorithms.

Results: GPS vs. DQN

The results of McNemar's test comparing GPS (Algorithm A) to DQN (Algorithm B) across various maze configurations are summarized in Table 13. The Average Success Rate (ASR) reported in the table for each algorithm is based on Table 2:

- ASR (GPS) = $(a + b)/(a + b + c + d)$
- ASR (DQN) = $(a + c)/(a + b + c + d)$

All comparisons in Table 13 yield p-values substantially less than 0.05, demonstrating statistically significant improvements of GPS over DQN across all tested maze environments. The consistent outcomes and significant p-values robustly support the conclusion that the GPS method offers superior performance compared to the DQN baseline under the specified experimental conditions.

Table 13: McNemar's Test Results for GPS vs. DQN. All p-values < 0.05 indicate a statistically significant difference in performance, favoring GPS in all listed cases. P-values reported as 0.0000 by the script are presented as < 0.0001.

Maze Type	ASR (GPS)	ASR (DQN)	McNemar Stat.	p-value
8x8	1.00	0.95	54.0000	$<$ **0.0001**
16x16	1.00	0.69	377.0000	$<$ **0.0001**
16x16_obs_15	0.96	0.85	22.1538	$<$ **0.0001**
16x16_obs_25	0.90	0.8	28.4462	$<$ **0.0001**
16x16_rooms	0.92	0.65	136.5329	$<$ **0.0001**
16x16_corridors	1.00	0.80	107.0000	$<$ **0.0001**
24x24	1.00	0.24	757.0000	$<$ **0.0001**
24x24_obs_15	0.91	0.15	760.0208	$<$ **0.0001**
24x24_obs_25	0.36	0.11	192.9627	$<$ **0.0001**

The following sections provide the detailed per-run summaries logged and the specific contingency tables used for McNemar's test for each maze configuration.

Maze: 8x8

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 1000, Successes: 946, Failures: 54, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	946 (a)	54 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 54.0000, p-value: $<$ **0.0001**

Maze: 16x16

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0

- *DQN (Algorithm B) Summary:* Total episodes: 1000, Successes: 623, Failures: 377, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	623 (a)	377 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 999

McNemar's Statistic: 342.0000, p-value: < **0.0001**

Maze: 16x16_obs_15

- *GPS (Algorithm A) Summary:* Total episodes: 210, Successes: 202, Failures: 8, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 210, Successes: 178, Failures: 32, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	177 (a)	25 (b)
	Failure	1 (c)	7 (d)

Common episodes for comparison: 210

McNemar's Statistic: 22.1538, p-value: < **0.0001**

Maze: 16x16_obs_25

- *GPS (Algorithm A) Summary:* Total episodes: 399, Successes: 360, Failures: 39, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 402, Successes: 320, Failures: 82, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	306 (a)	54 (b)
	Failure	11 (c)	28 (d)

Common episodes for comparison: 399

McNemar's Statistic: 28.4462, p-value: < **0.0001**

Maze: 16x16_rooms

- *GPS (Algorithm A) Summary:* Total episodes: 586, Successes: 530, Failures: 56, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 586, Successes: 379, Failures: 207, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	371 (a)	159 (b)
	Failure	8 (c)	48 (d)

Common episodes for comparison: 583

McNemar's Statistic: 132.7872, p-value: < **0.0001**

Maze: 16x16_corridors

- *GPS (Algorithm A) Summary:* Total episodes: 545, Successes: 545, Failures: 0, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 545, Successes: 438, Failures: 107, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	438 (a)	107 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 545

McNemar's Statistic: 107.0000, p-value: < **0.0001**

Maze: 24x24

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 999, Failures: 1, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 1000, Successes: 242, Failures: 758, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	242 (a)	757 (b)
	Failure	0 (c)	1 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 757.0000, p-value: < **0.0001**

Maze: 24x24_obs_15

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 914, Failures: 86, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 1000, Successes: 150, Failures: 850, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	148 (a)	766 (b)
	Failure	2 (c)	84 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 760.0208, p-value: < **0.0001**

Maze: 24x24_obs_25

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 368, Failures: 632, Errors: 0
- *DQN (Algorithm B) Summary:* Total episodes: 1000, Successes: 99, Failures: 901, Errors: 0

Contingency Table (GPS vs. DQN):

		DQN (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	46 (a)	322 (b)
	Failure	53 (c)	579 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 192.9627, p-value: $<$ **0.0001**

This detailed breakdown for each environment shows the specific data underlying the McNemar's tests.

Results: GPS vs. TempoRL

Table 14 summarizes the Average Success Rates (ASR) for GPS and TempoRL, along with the McNemar test statistics and p-values derived from common paired evaluation episodes.

All comparisons in Table 14 yield p-values substantially less than 0.05, demonstrating statistically significant improvements of GPS over the TempoRL baseline across all tested maze environments. The consistent outcomes and significant p-values robustly support the conclusion that the GPS method offers superior performance compared to the TempoRL baseline under these experimental conditions.

Table 14: Summary of McNemar's Test Results for GPS vs. TempoRL. All p-values < 0.05 indicate a statistically significant difference in performance, favoring GPS.

Maze Type	**ASR (GPS)**	**ASR (TempoRL)**	**McNemar Stat.**	**p-value**
8x8	1.00	0.97	26.0000	$<$ **0.0001**
16x16	1.00	0.84	178.0000	$<$ **0.0001**
16x16_obs_15	0.96	0.82	19.5652	$<$ **0.0001**
16x16_obs_25	0.90	0.79	18.9804	$<$ **0.0001**
16x16_rooms	0.92	0.63	119.0088	$<$ **0.0001**
16x16_corridors	1.00	0.9	56.0000	$<$ **0.0001**
24x24	1.00	0.46	511.0078	$<$ **0.0001**
24x24_obs_15	0.91	0.20	651.0968	$<$ **0.0001**
24x24_obs_25	0.36	0.09	171.6100	$<$ **0.0001**

The following sections provide the detailed per-run summaries and the specific contingency tables used for McNemar's test for each maze configuration when comparing GPS with TempoRL.

Maze: 8x8 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 1000, Successes: 974, Failures: 26, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	974 (a)	26 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 26.0000, p-value: $<$ **0.0001**

Maze: 16x16 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 1000, Successes: 822, Failures: 178, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	822 (a)	178 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 178.0000, p-value: < **0.0001**

Maze: 16x16_obs_15 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 210, Successes: 202, Failures: 8, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 210, Successes: 172, Failures: 38, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	164 (a)	38 (b)
	Failure	8 (c)	0 (d)

Common episodes for comparison: 210

McNemar's Statistic: 19.5652, p-value: < **0.0001**

Maze: 16x16_obs_25 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 399, Successes: 360, Failures: 39, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 402, Successes: 318, Failures: 84, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	287 (a)	73 (b)
	Failure	29 (c)	10 (d)

Common episodes for comparison: 399

McNemar's Statistic: 18.9804, p-value: < **0.0001**

Maze: 16x16_rooms (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 586, Successes: 530, Failures: 56, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 586, Successes: 366, Failures: 220, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	335 (a)	195 (b)
	Failure	31 (c)	25 (d)

Common episodes for comparison: 586

McNemar's Statistic: 119.0088, p-value: $<$ **0.0001**

Maze: 16x16_corridors (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 545, Successes: 545, Failures: 0, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 545, Successes: 489, Failures: 56, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	489 (a)	56 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 545

McNemar's Statistic: 56.0000, p-value: $<$ **0.0001**

Maze: 24x24 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 999, Failures: 1, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 1000, Successes: 486, Failures: 514, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	485 (a)	514 (b)
	Failure	1 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 511.0078, p-value: $<$ **0.0001**

Maze: 24x24_obs_15 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 914, Failures: 86, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 1000, Successes: 218, Failures: 782, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	194 (a)	720 (b)
	Failure	24 (c)	62 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 651.0968, p-value: $<$ **0.0001**

Maze: 24x24_obs_25 (GPS vs. TempoRL)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 368, Failures: 632, Errors: 0
- *TemporL (Algorithm B) Summary:* Total episodes: 1000, Successes: 106, Failures: 894, Errors: 0

Contingency Table (GPS vs. TemporL):

		TemporL (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	37 (a)	331 (b)
	Failure	69 (c)	563 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 171.6100, p-value: < **0.0001**

This detailed breakdown for each environment when comparing GPS to TempoRL shows the specific data underlying McNemar's tests.

Results: GPS vs. DAR

Table 15 summarizes the Average Success Rates (ASR) for GPS and DAR, along with the McNemar test statistics and p-values derived from common evaluation episodes.

Table 15: Summary of McNemar's Test Results for GPS vs. DAR. All p-values < 0.05 indicate a statistically significant difference in performance, favoring GPS.

Maze Type	ASR (GPS)	ASR (DAR)	McNemar Stat.	p-value
8x8	1.00	0.76	244.0000	< **0.0001**
16x16	1.00	0.61	394.0000	< **0.0001**
16x16_obs_15	0.96	0.64	57.8000	< **0.0001**
16x16_obs_25	0.90	0.14	298.2038	< **0.0001**
16x16_rooms	0.92	0.15	438.0800	< **0.0001**
16x16_corridors	1.00	0.61	213.0000	< **0.0001**
24x24	1.00	0.23	711.0000	< **0.0001**
24x24_obs_15	0.91	0.12	614.2257	< **0.0001**
24x24_obs_25	0.36	0.07	236.2798	< **0.0001**

All comparisons in Table 15 yield p-values substantially less than 0.05, demonstrating statistically significant improvements of GPS over the DAR baseline across all tested maze environments. The consistent outcomes and significant p-values robustly support the conclusion that GPS offers superior performance compared to the DAR baseline under these experimental conditions.

The following sections provide the detailed per-run summaries logged by the script and the specific contingency tables used for McNemar's test for each maze configuration when comparing GPS with DAR.

Maze: 8x8 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 1000, Successes: 756, Failures: 244, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	756 (a)	244 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 244.0000, p-value: $<$ **0.0001**

Maze: 16x16 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 1000, Failures: 0, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 1000, Successes: 606, Failures: 394, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	606 (a)	394 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 394.0000, p-value: $<$ **0.0001**

Maze: 16x16_obs_15 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 210, Successes: 202, Failures: 8, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 210, Successes: 134, Failures: 76, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	128 (a)	74 (b)
	Failure	6 (c)	2 (d)

Common episodes for comparison: 210

McNemar's Statistic: 57.8000, p-value: $<$ **0.0001**

Maze: 16x16_obs_25 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 399, Successes: 360, Failures: 39, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 402, Successes: 54, Failures: 348, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	50 (a)	310 (b)
	Failure	4 (c)	35 (d)

Common episodes for comparison: 399

McNemar's Statistic: 298.2038, p-value: $<$ **0.0001**

Maze: 16x16_rooms (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 586, Successes: 530, Failures: 56, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 586, Successes: 86, Failures: 500, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	83 (a)	447 (b)
	Failure	3 (c)	53 (d)

Common episodes for comparison: 586
McNemar's Statistic: 438.0800, p-value: < **0.0001**

Maze: 16x16_corridors (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 545, Successes: 545, Failures: 0, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 544, Successes: 331, Failures: 213, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	331 (a)	213 (b)
	Failure	0 (c)	0 (d)

Common episodes for comparison: 544
McNemar's Statistic: 213.0000, p-value: < **0.0001**

Maze: 24x24 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 999, Failures: 1, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 1000, Successes: 288, Failures: 712, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	288 (a)	711 (b)
	Failure	0 (c)	1 (d)

Common episodes for comparison: 1000
McNemar's Statistic: 711.0000, p-value: < **0.0001**

Maze: 24x24_obs_15 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 914, Failures: 86, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 4833, Successes: 524, Failures: 4309, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	38 (a)	632 (b)
	Failure	6 (c)	55 (d)

Common episodes for comparison: 731

McNemar's Statistic: 614.2257, p-value: < **0.0001**

Maze: 24x24_obs_25 (GPS vs. DAR)

- *GPS (Algorithm A) Summary:* Total episodes: 1000, Successes: 368, Failures: 632, Errors: 0
- *DAR (Algorithm B) Summary:* Total episodes: 1000, Successes: 66, Failures: 934, Errors: 0

Contingency Table (GPS vs. DAR):

		DAR (Algorithm B)	
		Success	**Failure**
GPS (Alg. A)	*Success*	24 (a)	344 (b)
	Failure	42 (c)	590 (d)

Common episodes for comparison: 1000

McNemar's Statistic: 236.2798, p-value: < **0.0001**

This detailed breakdown for each environment when comparing GPS to DAR shows the specific data underlying McNemar's tests.