Thesis for Bachelor's Degree

Diffusion-Based Offline RL for Improved Decision-Making in Augmented ARC Task

Yunho Kim

School of Electrical Engineering and Computer Science

Gwangju Institute of Science and Technology

2024

학사학위논문

증강된 추상화와 추론 과제에서 확산 모델 기반 오프라인 강화학습을 이용한 의사결정 능력 향상

김윤호

전기전자컴퓨터공학부

광주과학기술원

2024

Diffusion-Based Offline RL for Improved Decision-Making in Augmented ARC Task

Advisor: Sundong Kim

by

Yunho Kim

School of Electrical Engineering and Computer Science

Gwangju Institute of Science and Technology

A thesis submitted to the faculty of the Gwangju Institute of Science and Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in the Electrical Engineering and Computer Science Concentration

Gwangju, Republic of Korea

Dec 06, 2024

Approved by

Professor Sundong Kim

Committee Chair

Diffusion-Based Offline RL for Improved Decision-Making in Augmented ARC Task

Yunho Kim

Accepted in partial fulfillment of the requirements for

the degree of Bachelor of Science

Dec 06, 2024

Committee Chair _____

Prof. Sundong Kim

Committee Member _____

Prof. Kangil Kim

BS/EC 20195036 Yunho Kim. Diffusion-Based Offline RL for Improved Decision-Making in Augmented ARC Task. School of Electrical Engineering and Computer Science. 2024. 25p. Advisor: Prof. Sundong Kim.

Abstract

Effective long-term strategies enable AI systems to navigate complex environments by making sequential decisions over extended horizons. Similarly, reinforcement learning (RL) agents optimize decisions across sequences to maximize rewards, even without immediate feedback. To verify that Latent Diffusion-Constrained Q-learning (LDCQ), a prominent diffusion-based offline RL method, demonstrates strong reasoning abilities in multi-step decision-making, I aimed to evaluate its performance on the Abstraction and Reasoning Corpus (ARC). However, applying offline RL methodologies to enhance strategic reasoning in AI for solving tasks in ARC is challenging due to the lack of sufficient experience data in the ARC training set. To address this limitation, I introduce an augmented offline RL dataset for ARC, called Synthesized Offline Learning Data for Abstraction and Reasoning (SOLAR), along with the SOLAR-Generator, which generates diverse trajectory data based on predefined rules. SOLAR enables the application of offline RL methods by offering sufficient experience data. I synthesized SOLAR for a simple task and used it to train an agent with the LDCQ method. Our experiments demonstrate the effectiveness of the offline RL approach on a simple ARC task, showing the agent's ability to make multi-step sequential decisions and correctly identify answer states. These results highlight the potential of the offline RL approach to enhance AI's strategic reasoning capabilities.

BS/EC 20195036 김윤호. 증강된 추상화와 추론 과제에서 확산 모델 기반 오프라인 강화학습을 이용한 의사결정 능력 향상. 전기전자컴퓨터공학부. 2024. 25p. 지도교수: 김선동.

국 문 요 약

효과적인 장기 전략은 AI 시스템이 복잡한 환경을 순차적 결정을 통해 장기적으로 탐색할 수 있게 한다. 이와 유사하게, 강화 학습 (RL) 에이전트는 즉각적인 피드백 없이도 보상을 극대화하기 위해 연속적인 결정을 최적화한다. 본 연구는 대표적인 확산 기반 오프라인 RL 방법인 Latent Diffusion-Constrained Q-learning (LDCQ)가 다단계 의사결정에서 강력한 추론 능력을 발휘하는지 검증하기 위해 ARC (Abstraction and Reasoning Corpus)에서 그 성능을 평가하는 것을 목표로 힌다. 그러나 ARC 훈련 세트에 충분한 경험 데이터가 부족하여, 오프라인 RL 방법론을 ARC 과제 해결을 위한 AI의 전략적 추론 강화에 적용하는 데 어려움이 있다. 이러한 한계를 해결하기 위해, ARC를 위한 증강 오프라인 RL 데이터셋인 Synthesized Offline Learning Data for Abstraction and Reasoning (SOLAR)와 규정된 규칙을 기반으로 다양한 경로 데이터를 생성하는 SOLAR-Generator를 소개한다. SOLAR는 충분한 경험 데이터를 제공하여 오프라인 RL 방법을 적용할 수 있게 한다. 간단한 과제를 위해 SOLAR 데이터를 생성하고 이를 사용하여 오프라인 강화학습 방법 중 하나인 LDCQ 방법으로 에이전트를 훈련시켰다. 실험 결과, 오프라인 RL 접근 방식이 간단한 ARC 과제에서 효과적임을 보여주었으며, 에이전트가 다단계 순차적 결정을 내리고 정답 상태를 올바르게 식별할 수 있음을 확인했다. 이러한 결과는 오프라인 RL 접근 방식이 AI의 전략적 추론 능력을 향상할 가능성을 시사한다.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1
Introduction

Effective long-term strategies involve deliberate reasoning, which refers to the thoughtful evaluation of options to determine the best course of action [1]. This type of reasoning requires conscious effort and allows intelligent beings to systematically plan and execute multi-step strategies to achieve complex long-term goals. Similarly, reinforcement learning (RL) agents make decisions with the goal of maximizing rewards over extended sequences of actions, even without immediate feedback. In both cases, reasoning involves considering a sequence of actions to reach an optimal outcome. The way Q-values guide an RL agent toward desired outcomes can be seen as aligning with the subgoals of deliberate reasoning, particularly in terms of multi-step decision-making to achieve long-term objectives.

Recent approaches to offline RL combined with generative diffusion models have shown significant improvements in multi-step strategic decision-making abilities for future outcomes [2, 3, 4, 5]. In particular, Latent Diffusion-Constrained Q-learning (LDCQ) [6] leverages diffusion models to sample various latents that compress multi-step trajectories. These latents are then used to guide the Q-learning process. By generating diverse data based on in-distribution samples, diffusion models help overcome the limitations of fixed datasets. This integration of diffusion models into offline RL enhances agents' reasoning abilities, allowing them to consider multiple plausible trajectories across extended sequences.

This research aims to apply the offline RL method to tackle reasoning benchmarks that demand advanced reasoning capabilities. To this end, we chose the Abstraction and Reasoning Corpus (ARC) [7], one of the key benchmarks for measuring abstract reasoning ability in AI. As shown in Figure 1.1, the ARC training set consists of 400 grid-based tasks, each requiring the identification of common rules from the demon-

stration examples, which are then applied to solve the test examples. ARC tasks are particularly challenging for AI models because they require high-level reasoning abilities, integrating core knowledge priors such as objectness, basic geometry, and topology [7]. These core knowledge priors guide the decision-making process for selecting the appropriate actions. Therefore, I believe that agents trained with offline RL methods can leverage these core knowledge priors by learning from experienced data.

However, the existing ARC training dataset lacks sufficient trajectories to train agents with offline RL methods. To address this limitation, this research proposes Synthesized Offline Learning data for Abstraction and Reasoning (SOLAR), a dataset for training offline RL agents. SOLAR provides diverse trajectory data, allowing the agent to encounter various actions shaped by the core knowledge priors across different episodes. In this research, I generated SOLAR for a simple task using the SOLAR-Generator, which was then used to train agents with the LDCQ method.

This research attempts to apply offline RL methods to solving ARC tasks. Training with LDCQ on SOLAR enabled agents to devise pathways to correct answer states, including solution paths not present in the training data. This demonstrates the potential of diffusion-based offline RL to enhance AI's reasoning capabilities.

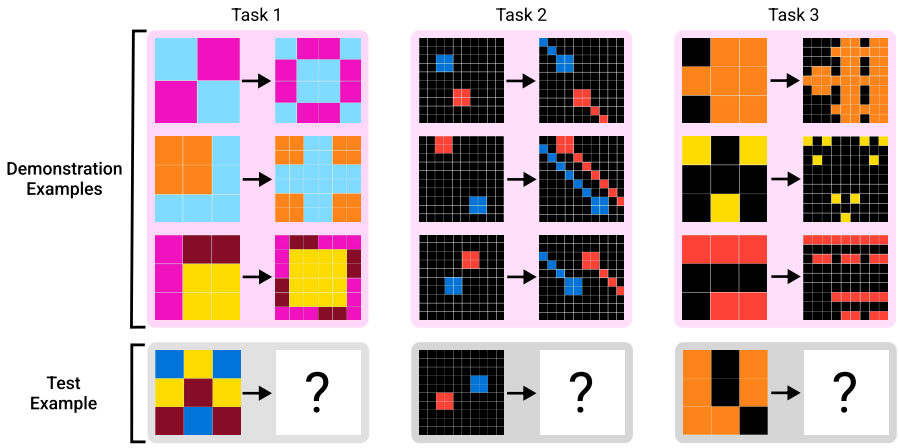


Figure 1.1: Three tasks in ARC. Each task consists of demonstration examples and a test example. Each example has an input grid and an output answer grid. Each pixel in the grid is matched to a color corresponding to a value in the range 0–9. ARC requires identifying common rules from the demonstration examples and applying them to solve the test example correctly. Despite recent advancements in AI, current models have consistently underperformed compared to humans on the ARC benchmark [8, 9].

Chapter 2

Preliminaries

2.1 ARC Learning Environment (ARCLE)

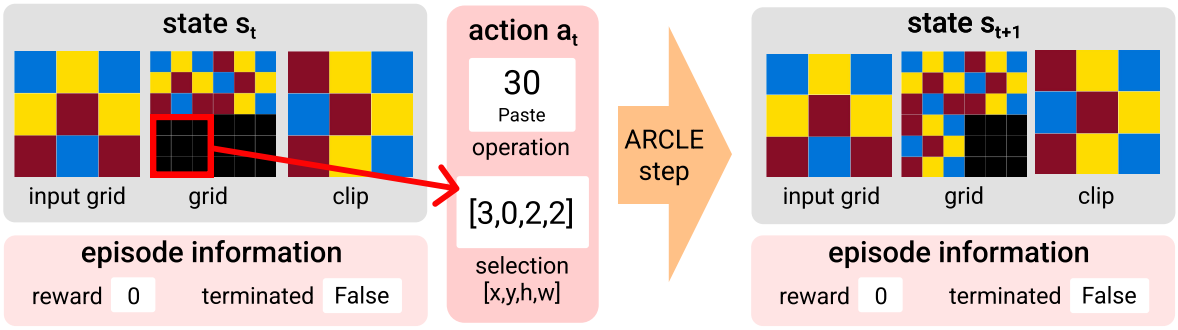


Figure 2.1: An example of a single step in ARCLE. In this example step, the action has an operation 30 (`Paste`) and a selection of $[3, 0, 2, 2]$. The top-left coordinate of the selection box is $[3, 0]$ and the bottom-right coordinate is $[5, 2]$. $[h_t, w_t]$ is calculated by subtracting $[3, 0]$ from $[5, 2]$. When ARCLE executes this action, the current clipboard is pasted into the bounding box specified by the selection on the current grid. It then returns episode information, including the reward and termination status.

ARCLE [10] is a Gymnasium-based environment developed to facilitate RL approaches for solving ARC tasks. ARCLE frames ARC tasks within a Markov Decision Process (MDP) structure, providing an environment where agents can interact with and manipulate grid-based tasks. This MDP structure enables ARC tasks to be solved through sequential decision-making.

ARCLE handles states and actions following the O2ARC web interface [11]. As shown in Figure 2.1, when ARCLE executes an action $\boldsymbol{a}_t$ on the current state $\boldsymbol{s}_t$, it returns the next state $\boldsymbol{s}_{t+1}$, along with episode information about the reward and termination status. A state $\boldsymbol{s}_t$ consists of $(\text{input grid}, \text{grid}_t, \text{clipboard}_t)$ at timestep t. The input grid represents the initial state of the test example, the grid_t denotes the current grid at time t after several actions have been applied, and the clipboard_t stores the copied grid by the `Copy` operation. An action $\boldsymbol{a}_t$ consists of $(\text{operation}_t, x_t, y_t, h_t, w_t)$,

where operation$_t$ represents the type of transformation, x_t and y_t denote the coordinates of the top-left point of the selection box, and h_t and w_t represent the difference between the bottom-right and top-left coordinates. All subsequent notations for $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$ will adhere to this definition for clarity. Reward is only given when the `Submit` operation is executed at the answer state, and the episode terminates either after receiving the reward or when `Submit` is executed across multiple trials. All possible operations are mentioned in Appendix B.1.

2.2 Diffusion-Based Offline Reinforcement Learning

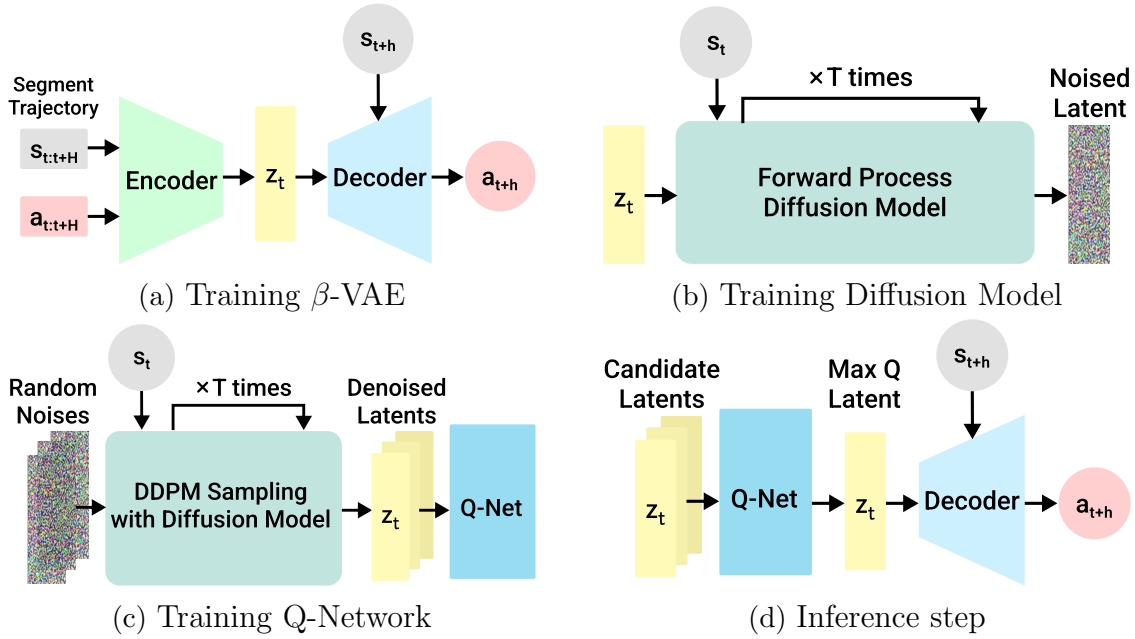
Offline RL focuses on learning policies from previously collected data, without interacting with the environment. However, Offline RL faces challenges, including data distribution shifts, limited diversity in the collected data, and the risk of overfitting to biased or insufficiently representative samples. To address these issues, several works in offline RL have focused on improving learning efficiency with large datasets and enhancing generalization to unseen scenarios while balancing diversity and ensuring data quality [12, 13, 14].

Recent offline RL methods offer promising solutions in long-horizon tasks and handling out-of-support samples through diffusion models. For instance, Diffuser [2] generates tailored trajectories by learning trajectory distributions, reducing compounding errors. Beyond this, a range of advanced diffusion-based offline RL approaches, such as Decision Diffuser (DD) [3], AdaptDiffuser [4], HDMI [5] have demonstrated the effectiveness of combining diffusion models with offline RL.

Chapter 3

LDCQ for ARC

Latent Diffusion-Constrained Q-learning (LDCQ) [6] combines latent diffusion and batch-constrained Q-learning to address long-horizon tasks with sparse rewards. By using sampled latents that encode H-length trajectories, LDCQ reduces extrapolation errors and enhances decision-making in multi-step tasks. The training process of LDCQ is shown in Figure 3.1: 1) training the β-VAE to learn latent representations, 2) training the diffusion model using the latent vectors encoded by the β-VAE, and 3) training the Q-network with latents sampled from the diffusion model.



(a) Training β-VAE

(b) Training Diffusion Model

(c) Training Q-Network

(d) Inference step

Figure 3.1: (a)–(c) Training stages of LDCQ. (a) Training a β-VAE with an encoder that encodes H-horizon segment trajectories into latents $\boldsymbol{z}_t$, and a policy decoder that decodes actions based on $\boldsymbol{z}_t$ and state $\boldsymbol{s}_{t+h}$ where $h \in [0, H)$ contained in the latent. (b) Training a diffusion model based on $\boldsymbol{z}_t$ and the $\boldsymbol{s}_t$. (c) Training a Q-network using latents sampled through the diffusion model. (d) LDCQ inference step at $\boldsymbol{s}_{t+h}$. Possible latents at $\boldsymbol{s}_t$ are sampled through the diffusion model, and the agent executes actions resulting from decoding the latent with the highest Q-value.

3.1 Training Latent Encoder and Policy Decoder

The first stage of LDCQ training is to train a β-VAE that learns latent representations. In this stage, β -VAE learns how actions are executed over multiple steps to change the state. With H-horizon latents, it becomes easier to capture longer-term changes in the state. I use SOLAR as the training dataset $\mathcal{D}$, which contains H-length segmented trajectories $\boldsymbol{\tau}_t$. Each $\boldsymbol{\tau}_t$ consists of state sequences $\boldsymbol{s}_{t:t+H} = [\boldsymbol{s}_t, \boldsymbol{s}_{t+1}, ..., \boldsymbol{s}_{t+H-1}]$ and action sequences $\boldsymbol{a}_{t:t+H} = [\boldsymbol{a}_t, \boldsymbol{a}_{t+1}, ..., \boldsymbol{a}_{t+H-1}]$, along with additional information such as demonstration examples. The state, as described in Section 2.1, consists of the test example represented as $(\text{input grid}, \text{grid}_t, \text{clipboard}_t)$. Each grid is passed through a layer to generate embeddings, which are then concatenated to form the state representation. Similarly, the action, as described in Section 2.1, is composed of $(\text{operation}_t, x_t, y_t, h_t, w_t)$, and each component is decoded individually by the policy decoder. The loss for action decoding is calculated as the sum of the losses for each component. The latent encoded by the β-VAE contains information about the ARC task's demonstration example, the test example input, the current state, and the action performed.

As shown in Figure 3.1a, during the β-VAE training stage, the encoder q_ϕ is trained to encode $\boldsymbol{\tau}_t$ into the latent representation $\boldsymbol{z}_t$, and the low-level policy decoder π_θ is trained to decode actions based on the given state and latent. For example, given the latent $\boldsymbol{z}_t$ and a state from the segment trajectory, $\boldsymbol{s}_{t+h}$ where $h \in [0, H)$, the policy decoder decodes the action $\boldsymbol{a}_{t+h}$ for $\boldsymbol{s}_{t+h}$. The β-VAE is trained by maximizing the evidence lower bound (ELBO), minimizing the loss in Eq. 3.1. The loss consists of the reconstruction loss from the low-level policy decoder and the KL divergence between the approximate posterior $q_\phi(\boldsymbol{z}_t|\boldsymbol{\tau}_t)$ and the prior $p_\omega(\boldsymbol{z}_t|\boldsymbol{s}_t)$.

$$\mathcal{L}_{\text{VAE}}(\theta, \phi, \omega) = -\mathbb{E}_{\boldsymbol{\tau}_t \sim \mathcal{D}} \left[\mathbb{E}_{q_\phi(\boldsymbol{z}_t|\boldsymbol{\tau}_t)} \left[\sum_{l=t}^{t+H-1} \log \pi_\theta(\boldsymbol{a}_l|\boldsymbol{s}_l, \boldsymbol{z}_t) \right] - \beta D_{KL}(q_\phi(\boldsymbol{z}_t|\boldsymbol{\tau}_t) \parallel p_\omega(\boldsymbol{z}_t|\boldsymbol{s}_t)) \right] \quad (3.1)$$

3.2 Training Latent Diffusion Model

In the second stage, the latent diffusion model is trained to generate latents based on the latent representations encoded by β -VAE. The training data consists of $(\boldsymbol{s}_t, \boldsymbol{z}_t)$ pairs, which are used to train a conditional latent diffusion model $p_\psi(\boldsymbol{z}_t|\boldsymbol{s}_t)$ by learning the denoising function $\mu_\psi(\boldsymbol{z}_t^j, \boldsymbol{s}_t, j)$, where $j \in [0, T]$ is diffusion timestep. This allows the model to capture the distribution of trajectory latents conditioned on $\boldsymbol{s}_t$. $q(\boldsymbol{z}_t^j|\boldsymbol{z}_t^0)$ denotes the forward Gaussian diffusion process that noising the original data. Following previous research [15, 6], the diffusion model is trained to predict the original latent rather than the noise, balancing the loss over diffusion time steps using the Min-SNR-γ strategy [16]. The loss function used to train the diffusion model is shown in Eq. 3.2. Here, $\boldsymbol{z}_t^j$, $j \in [0, T]$ represents noised latent on j-th diffusion time step, when $j = 0$ then $\boldsymbol{z}_t^0 = \boldsymbol{z}_t$ and $\boldsymbol{z}_t^T$ is Gaussian noise.

$$\mathcal{L}(\psi) = \mathbb{E}_{j\sim[1,T],\boldsymbol{\tau}_H\sim\mathcal{D},\boldsymbol{z}_t\sim q_\phi(\boldsymbol{z}_t|\boldsymbol{\tau}_t),\boldsymbol{z}_t^j\sim q(\boldsymbol{z}_t^j|\boldsymbol{z}_t^0)}\left[\min\{\mathrm{SNR}(j), \gamma\}\|\boldsymbol{z}_t^0 - \mu_\psi(\boldsymbol{z}_t^j, \boldsymbol{s}_t, j)\|^2\right] \quad (3.2)$$

With the trained diffusion model, diverse latents encapsulating candidate trajectories conditioned on the current state $\boldsymbol{s}_t$ can be sampled. These latents resemble trajectories in the training data while offering flexibility to generate plausible alternatives, allowing the model to generalize and evaluate multiple options before selecting an action. This enhances decision-making in unseen scenarios and is particularly beneficial for tasks with sparse rewards or ambiguous intermediate states, where diverse trajectories significantly improve reasoning capabilities.

3.3 Training Q-Network

Finally, the latent vectors sampled by the latent diffusion model are used for Q-learning. For sampling latents, I use the DDPM method [17]. The trained diffusion model samples latents by denoising random noise using the starting state information $\boldsymbol{s}_t$. A set of $(\boldsymbol{s}_t, \boldsymbol{z}_t, r_{t:t+H}, \boldsymbol{s}_{t+H})$ is used for training Q-Network, where $r_{t:t+H} =$

$\sum_{l=t}^{t+H-1} \gamma^l r_l$ denotes the discounted sum of rewards. Here, DDPM sampling is used to sample $\boldsymbol{z}_{t+H}$ for $\boldsymbol{s}_{t+H}$. For Q-learning, I use Clipped Double Q-learning [18] as shown in Eq. 3.3 with Prioritized Experience Replay buffer [19] to improve learning stability and mitigate overestimation. The trained Q-network $Q(\boldsymbol{s}_t, \boldsymbol{z}_t)$ evaluates the expected return of performing various H-length actions, with $\boldsymbol{z}_t$ sampled via DDPM based on $\boldsymbol{s}_t$. This allows the network to efficiently calculate the value of actions over H-steps to estimate future returns. Furthermore, since ARC tasks involve inferring analogies from demonstration pairs, the embedded representation of the demonstration pair, $\boldsymbol{p}_{emb}$, is also used in the calculation of the Q function. This approach aims to make the Q-values vary depending on the demonstration pair embeddings, encouraging the agent to consider the demonstration examples more effectively when selecting actions.

$$Q(\boldsymbol{s}_t, \boldsymbol{z}_t, \boldsymbol{p}_{emb}) \leftarrow \left(r_{t:t+H} + \gamma^H Q(\boldsymbol{s}_{t+H}, \underset{\boldsymbol{z} \sim p_\psi(\boldsymbol{z}_{t+H} | \boldsymbol{s}_{t+H})}{\operatorname{argmax}} Q(\boldsymbol{s}_{t+H}, \boldsymbol{z}, \boldsymbol{p}_{emb}), \boldsymbol{p}_{emb}) \right) \quad (3.3)$$

The detailed hyperparameters used for training the model are described in Appendix A.

Chapter 4
Synthesized Offline Learning data for Abstraction and Reasoning (SOLAR)

This research introduces a new dataset, Synthesized Offline Learning Data for Abstraction and Reasoning (SOLAR), which can be used to train offline RL methods. Solving ARC tasks can be considered a process of making multi-step decisions to transform the input grid into the output answer grid. I believe that the process of making these decisions inherently involves applying core knowledge priors, objectness, goal-directedness, numbers and counting, and basic geometry and topology [7], which are necessary for solving ARC tasks. The ARC training set lacks information on how to solve the task, and it only provides a set of demonstration examples and a test example for each task, as shown in Figure 1.1. To address this, I aim to provide the trajectory data to solve the task through SOLAR, enabling them to learn how actions change the state based on the application of core knowledge priors.

4.1 SOLAR Structure

SOLAR contains various transition data $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})$, where actions $\boldsymbol{a}_t$ are taken in different states $\boldsymbol{s}_t$, and the result $\boldsymbol{s}_{t+1}$ observed. To facilitate effective learning and a combination of core knowledge, I use ARCLE [10]. By designing a simple reward system that only provides rewards upon reaching the correct solution, I can guide the agent towards the desired state using reinforcement learning methods.

As shown in Figure 4.1, SOLAR consists of two key components: *Demonstration Examples* and *Test Example with Trajectory*. The demonstration examples and the test examples serve the same roles as in ARC. Through the demonstration examples, the common rule for transforming the input grid to the output grid is identified and then

applied to solve the test example. Trajectory data means the episode data that starts from test input s_0.

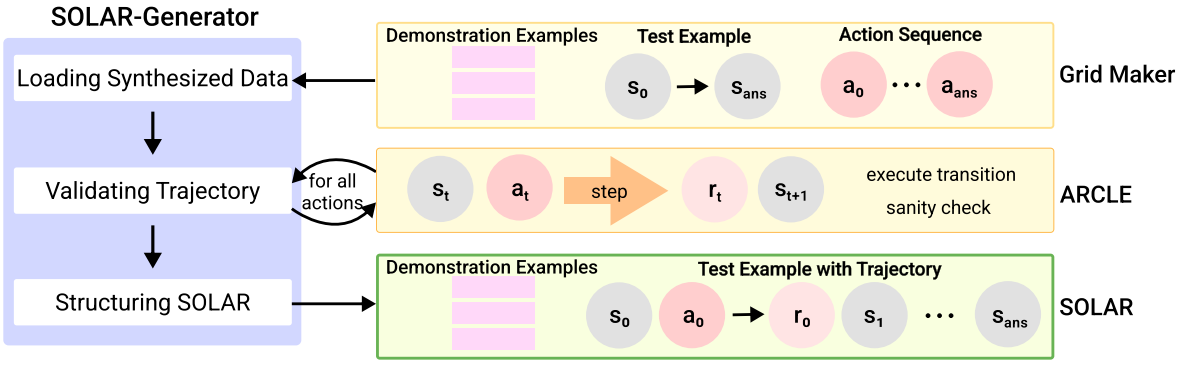


Figure 4.1: Data synthesis procedure with SOLAR-Generator. The state and actions consist of as mentioned in Section 2.1. 1) Loading Synthesized Data: The Grid Maker module applies constraints, augments input-output pairs, and synthesizes solutions for specific tasks by utilizing actions. 2) Validating Trajectories: Checks whether the generated actions are executable in ARCLE. 3) Structuring SOLAR: Organizes and stores the synthesized data in SOLAR based on the defined format. This step determines what information to include in the dataset and whether to segment episodes into fixed-length chunks or store them as a whole.

4.2 SOLAR-Generator

To synthesize SOLAR, I introduce SOLAR-Generator, which provides a method for generating diverse trajectory data. SOLAR-Generator augments ARC trajectories by following ARCLE formalism, addressing the inherent complexity and diversity of ARC tasks. Figure 4.1 illustrates the data synthesis procedure, which is carried out in three steps: 1) Loading Synthesized Data, 2) Validating Trajectories with ARCLE, and 3) Structuring SOLAR.

Loading Synthesized Data The first step in SOLAR-Generator is to load the synthesized data for the target tasks. SOLAR provides the Grid Maker with common parameters such as maximum grid size and the number of demonstration examples per test example. Each task has its own specific Grid Maker, which synthesizes demonstration examples, test examples, and corresponding action sequences (selections and

operations) based on the task's constraints and rules. If desired, non-optimal trajectories containing random actions can also be synthesized. At this stage, the Grid Maker synthesizes only grid pairs and possible action sequences. The full trajectory data for the test example is constructed after passing through ARCLE. More details about how the Grid Maker synthesizes the input-output grids and action sequences are described in Appendix B.

Validating Trajectories with ARCLE After synthesizing various grids and action sequences with the Grid Maker, SOLAR-Generator checks whether the action sequences are valid in ARCLE. The Grid Maker serves as a data loader, enabling it to load and validate the synthesized data. Through this process, ARCLE provides intermediate states, rewards, and termination status for each step, and verifies that each action is correctly executed in the current state. This step is particularly important for non-optimal trajectories, where operations and selections may be generated randomly, as invalid selections can sometimes be synthesized by the Grid Maker. For *gold standard* trajectories, intended as correct solutions, SOLAR-Generator ensures that the final grid of the trajectory matches the expected output grid of the test example. As a result, this stage is useful for checking and debugging the synthesized trajectories, preventing unintended errors.

Structuring SOLAR After the trajectory validation is complete, the episodes are saved into SOLAR. In this step, user can determine the necessary information to include in SOLAR. At its core, SOLAR includes episodes consisting of state, action, reward, and termination information at each step, which are essential for training with offline RL methods. In addition to the previously mentioned information, SOLAR can also store various data from ARCLE, such as grid sizes at each step, binary mask versions of selections, and other relevant information needed for different learning methods. In this research, I designed the data to work with methods like LDCQ, which require trajectories of fixed horizon length H. Therefore, the trajectories are segmented into fixed-length chunks with a horizon length of H.

Through these three steps, SOLAR-Generator synthesizes diverse solutions by altering action orders or using alternative operation combinations. This is achieved by the Grid Maker, which generates data using pre-implemented algorithms, enabling the user to create as many trajectories as needed. SOLAR provides a sufficient training set for learning various problem-solving strategies. By offering diverse trajectories while adhering to the task-solving criteria, SOLAR bridges the gap between ARC's reasoning challenges and the sequential decision-making process of offline RL. For additional details about SOLAR and SOLAR-Generator, see the project website[1] and Appendix B.

[1] `https://github.com/GIST-DSLab/SOLAR-Generator`

Chapter 5
Design SOLAR for a Simple Task

One of the most crucial factors in solving ARC tasks is the ability to recognize whether the current state is the answer state and to submit the correct answer accordingly. In ARC, each task embodies a single analogy, but this analogy can be approached through various action sequences [9, 20]. Some solution paths may better exemplify the underlying analogy, while others might be less optimal or clear [20]. Moreover, even when solving different test examples within the same task where the same rule is applied, the actual action sequence can vary depending on factors like the grid size or the arrangement of elements in the input grid. The diversity in potential action sequences to solve a single ARC task highlights the complexity of abstract reasoning and the importance of identifying the core analogy. Therefore, an agent's ability to judge that it has reached an answer state implies that it has comprehended the underlying analogy and executed the necessary ARCLE actions to arrive at the correct solution. This ability to recognize the answer state is critical, as it demonstrates the agent's understanding of the task's inherent logic and its capacity to apply appropriate problem-solving strategies. In ARCLE, the reward is given only when the agent predicts the `Submit` operation and the submitted grid is the same as the answer grid. To evaluate whether an agent trained with LDCQ can correctly identify and submit at the answer state—even when non-optimal trajectories are included in the training dataset—I mixed in incorrect episodes where the `Submit` operation is conducted in non-answer states.

Given these characteristics of ARC tasks, our experimental objectives are: 1) To assess whether the model can reach the answer state when various non-optimal trajectories are mixed with gold standard trajectories, and 2) To determine whether the model can recognize the answer state and perform the `Submit` action appropriately. By demonstrating the model's ability to identify the answer state, I can infer that it

has internalized core knowledge priors and understands the high-level problem-solving methods necessary for ARC tasks. I synthesized SOLAR for a simple task designed to show these experimental objectives.

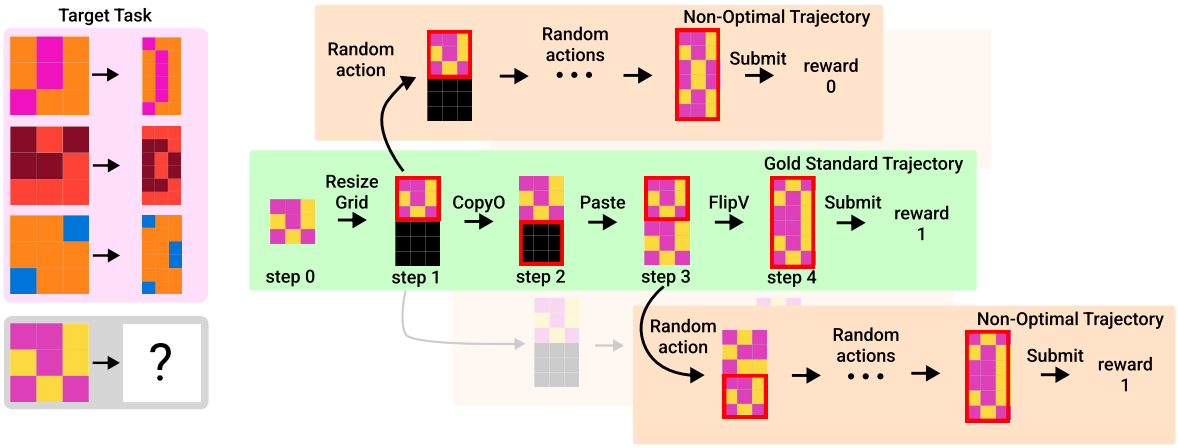


Figure 5.1: SOLAR episodes for a simple task: The gold standard trajectory (episode) contains the steps to solve the problem by using the core knowledge priors properly. The non-optimal episodes branch off at a random step within the standard trajectory, performing random operations such as `Rotate`, `Flip`, or `Copy` & `Paste`, and then `Submit` after a certain number of steps.

A simple task was designed to require core knowledge priors such as objectness and geometry. This task necessitates the ability to consider the input grid as an object and then perform actions based on this object. I constrained the maximum grid size to 10x10, and each episode includes three demonstration pairs. In creating SOLAR for this task, I constructed the dataset to include both gold standard episodes—which successfully reach the answer state and perform the `Submit` action—and non-optimal episodes—which follow random paths that may or may not reach the answer state. The inclusion of non-optimal trajectories was intended to evaluate whether the agent can recognize the answer state and appropriately perform the `Submit` action, thereby assessing its reasoning abilities rather than simply mimicking the actions in the dataset. As shown in Figure 5.1, the gold standard episode for this task consists of 5 steps: 1) `ResizeGrid` to make the grid two times longer vertically, 2) `CopyO` to copy the upper half of the current grid, as it matches the input grid, 3) `Paste` to apply it to the lower half of the grid, 4) `FlipV` to vertically flip the upper half of the current grid, and 5)

`Submit`, as it reaches the answer state.

In the non-optimal episodes, the trajectories initially follow the gold standard trajectory but deviate at a random step to execute random actions for several steps. I constrained the random operations to `FlipV` (vertical flip), `FlipH` (horizontal flip), `Rotate90` (counterclockwise rotation), `Rotate270` (clockwise rotation), and `Copy0` (updating the clipboard with the selected area). For selection, it was constrained to either two options (upper half or lower half of the current grid) or three options (upper half, lower half, or the whole grid). Specifically, there are two options for `Rotate90`, `Rotate270`, and `Copy0`, and three options for the others. When `Copy0` is selected, the subsequent `Paste` action is forced onto the other possible selection option. This simplified selection allows for focusing on assessing the AI's decision-making by sequentially combining operations.

Each non-optimal episode contains approximately ten steps to the end, allowing the trajectory to include various actions in diverse states. For each problem pair, one gold standard episode and nine non-optimal episodes were generated, totaling 5,000 episodes across 500 problem pairs. As a result, the training set was composed such that approximately 10% of the total episodes included the `Submit` operation at the answer state.

Chapter 6

Experiments and Results

6.1 Evaluation Process using ARCLE

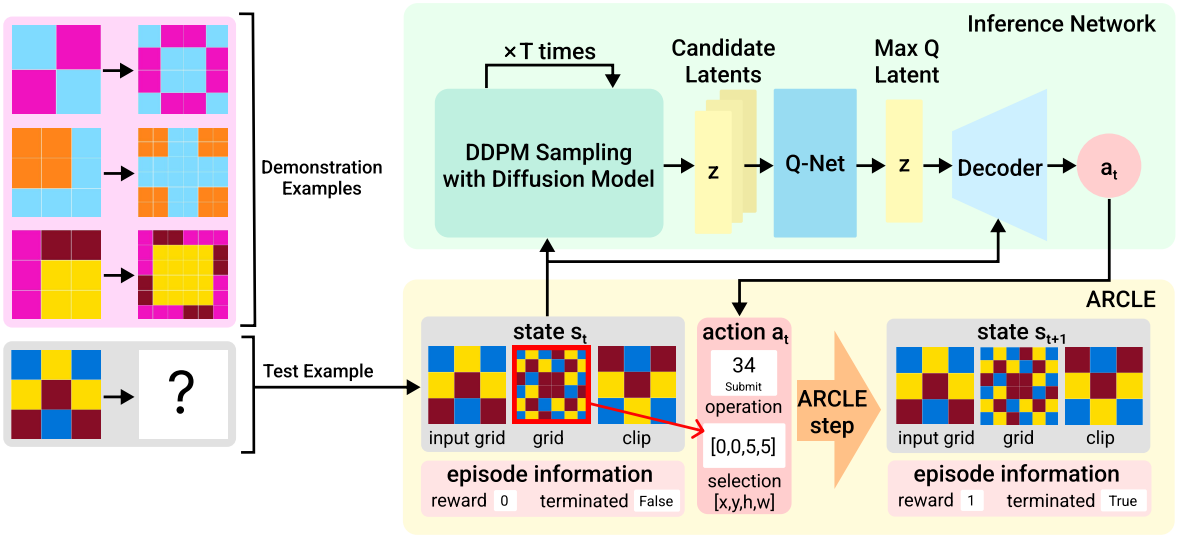


Figure 6.1: Inference framework for solving ARC tasks. ARCLE loads the task from the dataset and manages state information as well as the termination status of the current evaluation episode. The inference network of LDCQ performs DDPM sampling on the given state to extract candidate latents, then decodes the corresponding action for max Q latent, and sends it to ARCLE. ARCLE executes the action and updates the state information accordingly. This process alternates between ARCLE and the inference network, continuing the inference until the episode ends.

After training the agent using LDCQ on the SOLAR dataset, an evaluation of its performance was conducted. To evaluate the experiment, I synthesized an evaluation SOLAR set with 100 test examples, each paired with three synthesized demonstration examples. The evaluation SOLAR set was synthesized by the SOLAR-Generator using the same tasks but with a random seed different from the one used for the training set. To measure the effectiveness of decision-making using the Q-function, two accuracy metrics are measured: 1) Whether the agent reaches the answer state, and 2) Whether

it predicts the `Submit` operation at the answer state to receive a reward.

The evaluation process is carried out through ARCLE, which manages the problem and its corresponding solution from SOLAR. ARCLE handles state transitions, performs actions, and verifies whether the submitted solution is correct. As depicted in Figure 6.1, ARCLE interacts with the LDCQ inference network by alternating the exchange of $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, facilitating the decision-making process toward reaching the correct answer state. The latent $\boldsymbol{z}_t$ represents a segment trajectory spanning from timestep t to $t + H - 1$, and is trained to accurately decode actions for any state within this segment trajectory.

In the original LDCQ methodology, inference is performed by executing several horizons using a single latent, followed by predicting the next latent. However, in the task used for this research, which has a gold standard trajectory consisting of five steps, it is possible to complete the task with just one latent sampling from the initial state. While reaching the correct answer in this manner is not inherently problematic, one of the primary goals of this research is to analyze whether the agent learns the knowledge prior to how actions work across various states. Thus, instead of focusing solely on solving the problem in as few steps as possible, only one action is conducted per latent. With this, the results demonstrate that the agent can make far-sighted decisions to reach the answer not just from the beginning to the end, but also through intermediate steps.

6.2 Results

To demonstrate the strengths of the diffusion-based offline RL method guided by Q-function, I compare three approaches:

- **VAE prior (VAE)**: This method uses a latent sampled from the VAE state prior $p_\omega(\boldsymbol{z}_t|\boldsymbol{s}_t)$. The VAE state prior is trained in β-VAE training stage by calculating the KL divergence between $p_\omega(\boldsymbol{z}_t|\boldsymbol{s}_t)$ and the posterior $q_\phi(\boldsymbol{z}_t|\boldsymbol{\tau}_t)$, aligning the latent distribution with the trajectory starting from state $\boldsymbol{s}_t$.

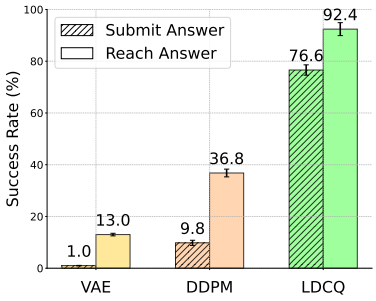
- **Diffusion prior (DDPM)**: This method uses a latent sampled from the diffusion model $p_\psi(\boldsymbol{z}_t|\boldsymbol{s}_t)$ through the DDPM method [17]. The sampled latents closely resemble the training data, with added variance during the denoising process. This method is similar to behavior cloning in that it operates without guidance from rewards or value functions.

- **Max Q latent (LDCQ)**: This method selects a latent with the highest Q-value from those sampled by the diffusion model, $\text{argmax}_{\boldsymbol{z}\sim p_\psi(\boldsymbol{z}_t|\boldsymbol{s}_t)} Q(\boldsymbol{s}_t, \boldsymbol{z})$, to make a decision at $\boldsymbol{s}_t$.

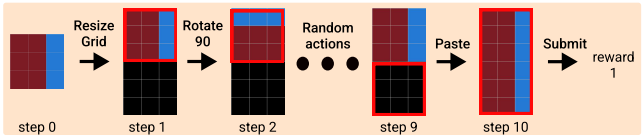
The evaluation of each approach was conducted five times for the evaluation SOLAR set. The results, summarized in Figure 6.2a, show the success rates for: 1) Whether the agent reaches the correct answer state and 2) Whether the agent executes `Submit` operation in the answer state. When using the VAE prior, the agent reaches the correct answer state in only about 10% of test episodes and submits the answer in just 1%. With latents sampled using DDPM, about 10% of the answers are correctly submitted, while the agent reaches the answer state approximately 37% of the time. When using LDCQ, the agent reaches the answer state in over 90% of cases and successfully submits the correct answer in about 77% of test episodes. These results demonstrate that the Q-function enhances the agent's ability to both reach the correct answer and recognize when it has arrived at the answer state.

Figure 6.2b and Figure 6.2c highlight the different solving strategies exhibited by the Q-function. When using the latent sampled with DDPM, the agent performs diverse actions, occasionally reaching the goal by chance. In contrast, with the Q-function, the agent consistently reaches the correct answer in every evaluation. In scenarios where the input grid is vertically symmetrical, the agent even skips unnecessary operation `FlipV` and proceeds directly to `Submit`. Notably, the training dataset does not include any trajectories where the `FlipV` operation is skipped, even for symmetrical grids. With the Q-function, the model recognizes that applying `FlipV` does not alter the state. Consequently, the Q-value for submitting at that state increases, prompting the agent

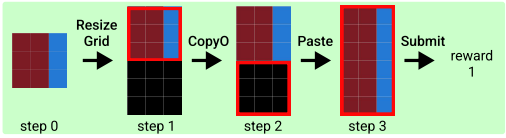
to choose the `Submit` operation. This demonstrates the reasoning ability of the agent trained with LDCQ in solving ARC tasks, as recognizing when the correct answer state has been reached is crucial.



(a) Test accuracy for three methods



(b) Inference example with DDPM method



(c) Inference example with LDCQ method

Figure 6.2: (a) The evaluation results for 100 test examples. LDCQ shows significantly improved performance compared to the other two methods, successfully reaching the correct answer state and executing the `Submit` operation at the answer state. The error bars represent the 96% confidence interval. (b) With the latent sampled with DDPM, the agent sometimes reaches the correct answer after performing various actions. This occurred rarely during evaluation, and even when it did, it did not appear in subsequent evaluations. (c) When using LDCQ, it often shows the case that skips unnecessary actions. The inference example with the VAE prior method is omitted because it rarely solves the problem.

Chapter 7
Limitations & Discussions

In our experiment, the LDCQ method showed significant improvement in reaching the goal. However, in approximately 16% of cases, the agent reached the correct state but proceeded with another action instead of submitting the solution, even with the assistance of the Q-function. This issue arises because the Q-function, while enhancing decision-making, sometimes assigns higher values to actions other than submission, causing the agent to bypass the goal state. This suggests that the Q-function is not perfectly aligned with the final objective in ARC. Notably, in ARC tasks, even when solving different test examples within the same task where the same rule is applied, the actual action sequence can vary depending on factors like grid size or the arrangement of elements in the input grid. The current Q-values are calculated based on the absolute state values, which occasionally leads to misjudgments when submitting the correct solution. Therefore, improving the agent's ability to accurately determine when to submit the correct answer is necessary for future research.

While the LDCQ approach performs well in a simple ARC task setting, more complex tasks and multi-task environments present additional challenges. Unlike single-task scenarios, where the agent follows a fixed strategy toward a predefined answer, multi-task settings demand flexibility to adapt to changing goals or new possibilities during task execution. I expect that addressing these challenges could involve integrating task classifiers for Q-learning. Additionally, incorporating modules so that the agent can revise its strategy during task execution—adjusting based on evolving states or objectives rather than rigidly following the initial strategy—may enhance its adaptability.

In traditional supervised RL approaches, such as those described by [21], stitching typically occurs only when the goal remains consistent across tasks. To address this limitation, I employed temporal data augmentation, which involves starting from an

intermediate state near the goal and setting a new target. In SOLAR, this could be extended by using non-optimal paths as goals in non-optimal trajectories. However, in ARC, where goals are determined by demonstration pairs, augmenting all goals is impractical. More careful strategies are needed to enable stitching for entirely new goals not previously encountered. If methodologies are developed that can combine existing actions toward different goals, I expect that SOLAR will facilitate these combinations.

Going forward, refining how the Q-function evaluates states and actions will be crucial. To improve performance, especially in multi-task environments, incorporating mechanisms that not only assess the state and action in relation to the goal but also guide the agent toward the most effective path to achieve the ultimate objective will be beneficial. Recognizing the task's context and how close states are to the correct solution is essential for ensuring that the Q-function helps navigate toward the goal efficiently.

Chapter 8
Conclusion

This research demonstrates the potential of offline reinforcement learning (RL), particularly the Latent Diffusion-Constrained Q-learning (LDCQ) method, for efficiently sequencing and organizing actions to solve tasks in grid-based environments like the Abstraction and Reasoning Corpus (ARC). To our knowledge, this work is the first to tackle ARC using a diffusion-based offline RL model within a properly designed environment, guiding agents step-by-step toward correct solutions without generating the full ARC grid at once. Through training on SOLAR, I successfully applied and evaluated offline RL methods, showing that agents can learn to find paths to the correct answer state and recognize when they've reached it. This suggests that RL with a well-designed environment is promising for abductive reasoning problems, potentially reducing data dependency compared to traditional methods. As tasks become more complex, especially in multi-task settings, refining the Q-function to address unique reward structures is crucial, with multi-task environments requiring task-specific adaptations to account for varying states and rewards. Integrating modules like task classifiers or object detectors could enhance the agent's ability to dynamically adjust its strategy, promoting more flexible decision-making. This research opens new avenues for program synthesis in analogical reasoning tasks with RL environments, potentially integrating with analogy findings techniques (hypothesis search with LLMs) to handle a wider range of ARC tasks.

References

1. D. Kahneman, *Thinking, Fast and Slow.* Farrar, Straus and Giroux, 2011.

2. M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine, "Planning with Diffusion for Flexible Behavior Synthesis," in *ICML*, 2022.

3. A. Ajay, Y. Du, A. Gupta, J. Tenenbaum, T. Jaakkola, and P. Agrawal, "Is Conditional Generative Modeling All You Need for Decision-Making?," in *ICLR*, 2023.

4. Z. Liang, Y. Mu, M. Ding, F. Ni, M. Tomizuka, and P. Luo, "AdaptDiffuser: Diffusion Models as Adaptive Self-Evolving Planners," in *ICML*, 2023.

5. W. Li, X. Wang, B. Jin, and H. Zha, "Hierarchical Diffusion for Offline Decision Making," in *ICML*, 2023.

6. S. Venkatraman, S. Khaitan, R. T. Akella, J. Dolan, J. Schneider, and G. Berseth, "Reasoning with Latent Diffusion in Offline Reinforcement Learning," in *ICLR*, 2024.

7. F. Chollet, "On the Measure of Intelligence," *arXiv:1911.01547*, 2019.

8. F. Chollet, M. Knoop, B. Landers, G. Kamradt, H. Jud, W. Reade, and A. Howard, "ARC Prize 2024," 2024.

9. A. Johnson, W. K. Vong, B. M. Lake, and T. M. Gureckis, "Fast and Flexible: Human Program Induction in Abstract Reasoning Tasks," in *CogSci*, 2021.

10. H. Lee, S. Kim, S. Lee, S. Hwang, J. Lee, B.-J. Lee, and S. Kim, "ARCLE: The Abstraction and Reasoning Corpus Learning Environment for Reinforcement Learning," in *CoLLAs*, 2024.

11. S. Shim, D. Ko, H. Lee, S. Lee, D. Song, S. Hwang, S. Kim, and S. Kim, "O2ARC 3.0: A Platform for Solving and Creating ARC Tasks," in *IJCAI*, 2024.

12. S. Fujimoto, D. Meger, and D. Precup, "Off-Policy Deep Reinforcement Learning without Exploration," in *ICML*, 2019.

13. R. Kidambi, A. Rajeswaran, P. Netrapalli, and T. Joachims, "MOReL: Model-Based Offline Reinforcement Learning," in *NeurIPS*, 2020.

14. S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems," *arXiv:2005.01643*, 2020.

15. A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical Text-Conditional Image Generation with CLIP Latents," *arXiv preprint arXiv:2204.06125*, 2022.

16. T. Hang, S. Gu, C. Li, J. Bao, D. Chen, H. Hu, X. Geng, and B. Guo, "Efficient Diffusion Training via Min-SNR Weighting Strategy," in *ICCV*, 2023.

17. J. Ho, A. Jain, and P. Abbeel, "Denoising Diffusion Probabilistic Models," in *NeurIPS*, 2020.

18. S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *ICML*, 2018.

19. T. Schaul, “Prioritized Experience Replay,” in *ICLR*, 2016.

20. S. Kim, P. Phunyaphibarn, D. Ahn, and S. Kim, “Playgrounds for Abstraction and Reasoning,” in *NeurIPS Workshop on Neuro Causal and Symbolic AI*, 2022.

21. R. Ghugare, M. Geist, G. Berseth, and B. Eysenbach, “Closing the Gap between TD Learning and Supervised Learning - A Generalisation Point of View,” in *ICLR*, 2024.

Appendix A
Training details

A.1 Hyperparameters

I used a horizon length of 5 for encoding skill latents, allowing the model to plan and evaluate actions over a five-step lookahead. The diffusion model was trained with 500 diffusion steps to minimize variance in the sampling process and ensure accurate decoding of operations and selections in ARCLE. The discount factor was set to 0.5 to balance immediate and future rewards, considering that ARCLE tasks typically require fewer than 20 steps to reach the correct answer.

The hyperparameters used for training the three stages of LDCQ are summarized in Tables A.1, A.2, and A.3.

Parameter	**Value**
Learning rate	5e-5
Batch size	128
Epochs	400
Horizon (H)	5
Latent dimension (z)	256
KL loss ratio (β)	0.1
Hidden layer dimension	512

Table A.1: Hyperparameters for training β-VAE

Parameter	Value
Learning rate	1e-4
Batch size	32
Epochs	400
Diffusion steps (T)	500
Drop probability	0.1
Variance schedule	linear
Sampling algorithm	DDPM
γ (For Min-SNR-γ weighing)	5

Table A.2: Hyperparameters for training latent diffusion model

Parameter	Value
Learning rate	5e-4
Batch size	128
Discount factor (γ)	0.5
Target net update rate (ρ)	0.995
PER buffer α	0.7
PER buffer β	Linearly increased from 0.3 to 1, grows by 0.03 every 2000 steps
Diffusion samples for batch argmax	100

Table A.3: Hyperparameters for training DQN

A.2 Hardware

I used an NVIDIA A100-SXM4-40GB GPU to train the model. Training the β-VAE took about 7 hours, while training the diffusion model and Q-network each took around 6 to 10 hours.

Appendix B

SOLAR-Generator

B.1 Operations in SOLAR

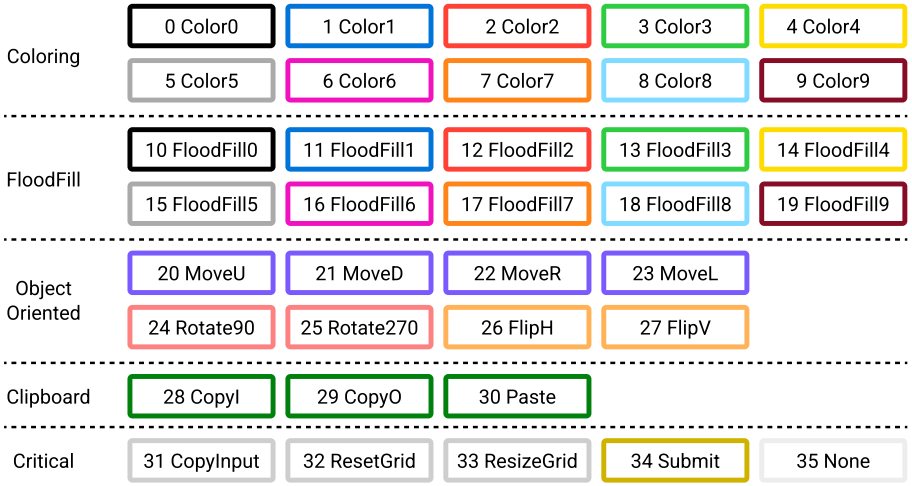


Figure B.1: All operations compatible with SOLAR, 0–34 operations follow ARCLE, and only in SOLAR, 35 (`None`) is for terminated episode. It means the episode is ended after `Submit`.

The operations from 0 to 34 are identical to those used in ARCLE [10]. Since `Submit` is an operation that receives a reward, it should only be used when the state is considered correct and not excessively. Due to LDCQ's fixed horizon, and to ensure that the agent only uses `Submit` when the state is definitively correct, we added a `None` operation that fills all subsequent states after `Submit` with the 11th color (10), which does not exist in the original ARC (0–9). In other words, during training, the `None` action emphasizes that the episode ends after `Submit`.

B.2 Detailed Procedure for Generating SOLAR

For generating SOLAR, we create a generator that can synthesize a large amount of data for a given rule. Grid Maker is a hard-coded program specific to each task.

Grid Maker contains the rules for synthesizing demonstration examples and test examples, and the synthesized solution action path consists of operations and selections. In Grid Maker, data is formatted to be compatible with ARCLE. The Grid Maker constructs analogies with the same problem semantics but with various attributes such as the shape, color, size, and position of objects. SOLAR-Generator can generate intermediate trajectories by interacting with ARCLE. The algorithm of the SOLAR-Generator is designed to augment specific tasks using the Grid Maker.

Grid Maker was built as a data loader, which is used in ARCLE. In the original ARCLE environment, there was no need to load operations and selections—only the grid was loaded since the problem alone was sufficient. To change this structure, the entire environment would need to be recreated. Instead, operations and selections are now loaded from the data loader's description, allowing us to retain the original environment. Therefore, the process of creating input-output examples and generating action sequences works within a single file. Grid Maker generates input-output examples and trajectories through the following three steps.

Specifying Common Parts Each task in the ARC dataset usually contains 3 demonstration examples, with common elements observed across these pairs. In the common parts, attributes such as color, the type of task, and the presence of objects are predetermined using random values before pair generation.

Synthesizing Examples In the example synthesis phase, the input of the original task is augmented in a way that ensures diversity while preserving the integrity of the problem-solving method. A random input grid is generated under conditions that satisfy the analogy required by the task. A solution grid is created using a hard-coded algorithm. For tasks involving pattern-based problems, as experimented in the paper, selections are made to fit the grid size, and various operations are executed either randomly or in a predetermined order. For object-based problems, the solution grid is generated by an algorithm that finds the necessary objects in the input grid and processes them according to the task requirements.

Converting to ARCLE Trajectories This stage involves the creation of an ARCLE-based trajectory that meticulously adheres to the problem-solving schema of the synthesized examples. The entire process is carried out through a hard-coded algorithm. During the example synthesis process, the locations of objects may already be known, or they can be identified using a search algorithm. The information obtained is then used to make the appropriate selections, and the trajectory is converted into an ARCLE trajectory through an algorithm that leads to the correct solution.

If all steps are properly coded, it is possible to generate the operations and selections that lead to the correct solution for any random input grid. These are then fed into ARCLE to obtain intermediate states, rewards, and other information, and to verify whether the correct result is reached. Once steps 1) to 3) are correctly implemented, SOLAR-Generator can continuously and automatically generate as much data for the given task as the user desires, using the Grid Maker.

B.3 Example of Data Synthesis in Grid Maker and the Generation of SOLAR

SOLAR-Generator can synthesize SOLAR for object-based tasks. Figure B.2 shows a variant of Task 2 from Figure 1.1. Grid Maker generates random input grids with some variances first. In this variant, each episode randomly selects two colors for the boxes. Each inputs can have different grid sizes, and rules are established for objects of each color within the episode. Then it generates the answer output grids for the input grids through algorithm. The solution algorithm in Grid Maker proceeds as follows: 1) Find the top-left corner of the orange square and repeat the coloring process to draw a diagonal line to the grid's edge. 2) Find the bottom-right corner of the red square and repeatedly color diagonally until the end of the grid is reached. With these algorithms, Grid Maker can synthesize as many examples and SOLAR trajectories as the user desires.

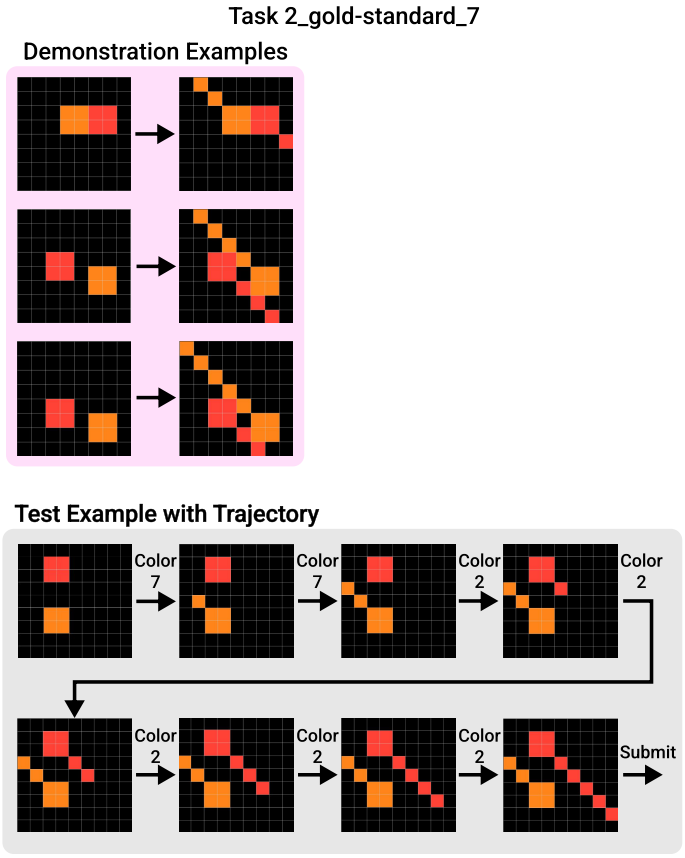


Figure B.2: A gold standard trajectory for Task 2 in Figure 1.1. SOLAR contains its trajectory ID, demonstration examples, and a test example with trajectory.

B.4 Algorithm of SOLAR-Generator

With the synthesized data through the Grid Maker module, the SOLAR-Generator checks the sanity of the synthesized trajectory, and then saves the data. The whole algorithm for SOLAR-Generator is described in Algorithm 1.

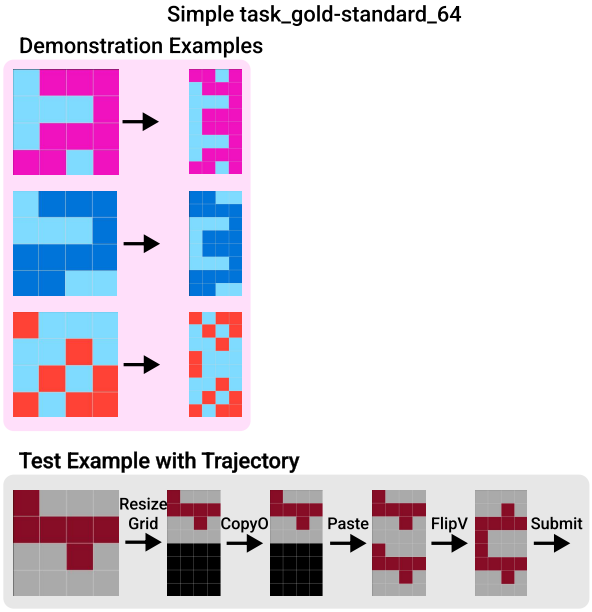
Algorithm 1: SOLAR-Generator

Input: task set T, maximum grid size (H, W), number of samples N, number of examples E

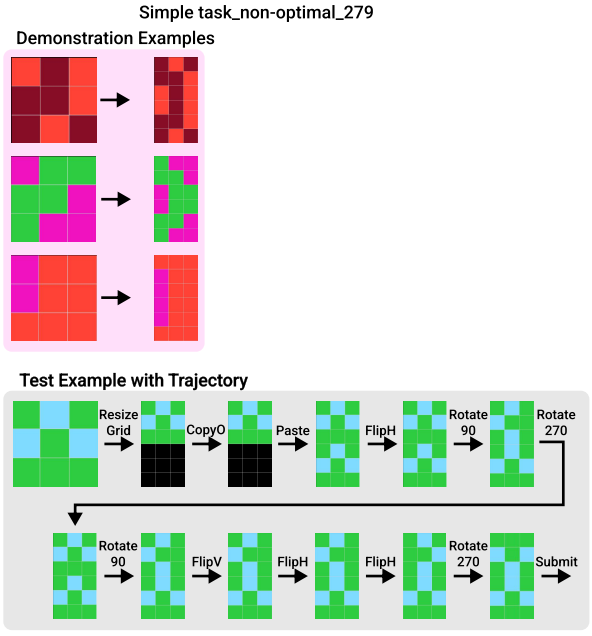
for $task \in T$ **do**
  # Load the synthesized data $\mathcal{D}_s$ from the Grid Maker for the *task*
  $\mathcal{D}_s \leftarrow$ Grid Maker(*task*, (H, W), N, E)
  for $data \in \mathcal{D}_s$ **do**
    # Extract the demonstration examples, test example, and actions for each episode
    trajectory_ID, dem_ex, input_grid, output_grid, operations, selections $\leftarrow$ *data*
    Add *trajectory_ID, dem_ex, input_grid, output_grid* to episode τ_{data}
    # Set the initial state
    $current\_grid_0 \leftarrow input\_grid$
    $clip\_grid_0 \leftarrow None$
    $t \leftarrow 0$
    $s_t \leftarrow (input\_grid, current\_grid_0, clip\_grid_0)$
    for $(opr_t, sel_t) \in$ *(operations, selections)* **do**
      $a_t \leftarrow (opr_t, sel_t)$
      if a_t can be performed in s_t **then**
        # Update state and episode information using ARCLE
        $current\_grid_{t+1}, clip\_grid_{t+1}, r_t, terminated_t \leftarrow$ `ARCLE.step`(s_t, a_t)
        Add $s_t, a_t, r_t, terminated_t$ to τ_{data}
        $s_{t+1} \leftarrow (input\_grid, current\_grid_{t+1}, clip\_grid_{t+1})$
        $t \leftarrow t + 1$
      else
        Save wrong data for debugging
        break
    if *"gold-standard"* in *trajectory_ID* and $current\_grid \neq output\_grid$ **then**
      Save wrong data for debugging
    else
      Save episode τ_{data}

B.5 Other SOLAR Examples

Figure B.3 illustrates two examples of episodes used in the experiment. Each episode includes three random demonstration examples and a trajectory for a test example.



(a) Gold standard episode



(b) Non-optimal episode

Figure B.3: SOLAR episode examples. (a) Gold standard episode that ideally reaches the answer. (b) Non-optimal episode that is not ideal, but still reaches the answer state.

Figure B.4 illustrates two different gold standard episodes. There might be multiple gold standard trajectories in the same test example.

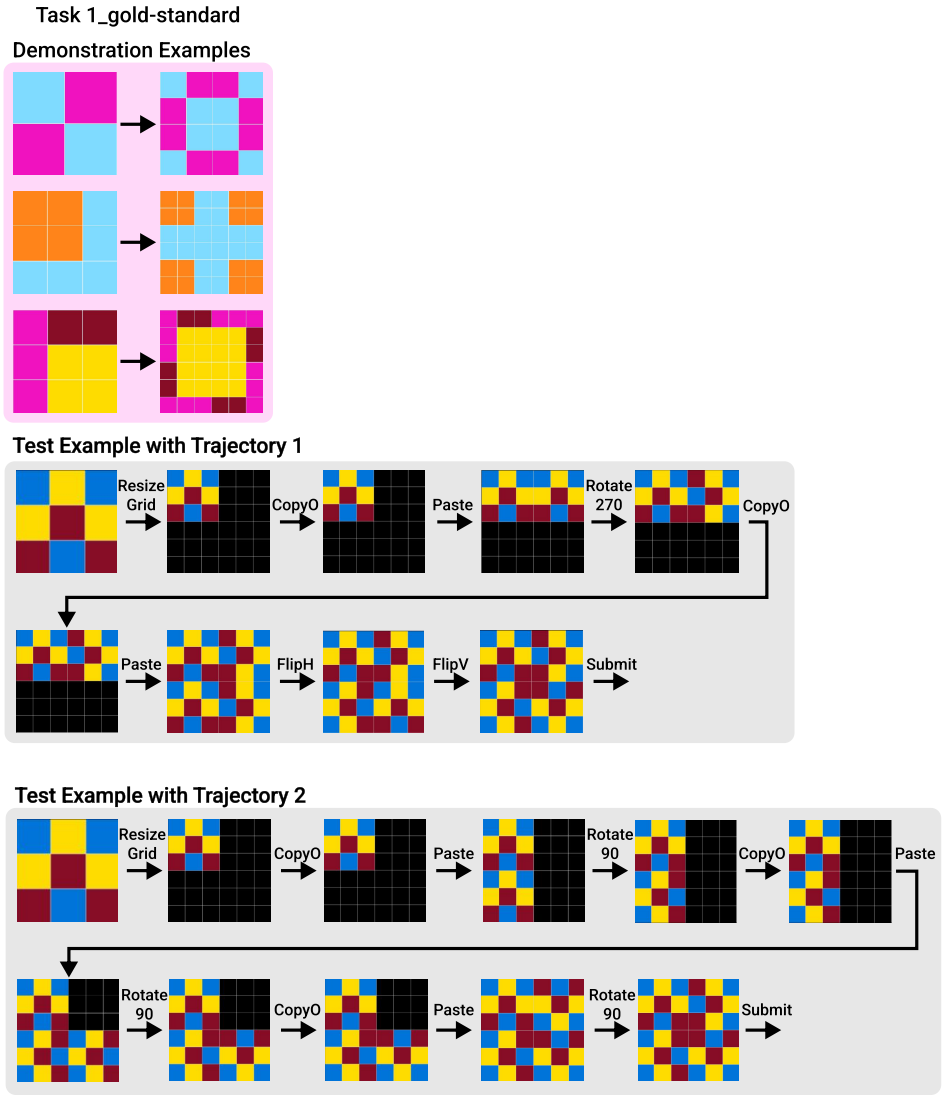


Figure B.4: Two different gold standard trajectories for Task 1 in Figure 1.1.