# Solving ARC with Deep Reinforcement Learning

A Novel Framework for Sequential Solutions

**Francesco Braicovich** Head
**Vittorio Garavelli** Member
**Filippo Gombac** Member
**Mariano Masiello** Member

Hephaestus Applied Artificial Intelligence Association

**Abstract**

We propose a reinforcement learning (RL) framework for solving tasks in the Abstraction and Reasoning Corpus (ARC) through sequential transformation prediction. Departing from existing domain-specific languages (DSLs) tailored for program synthesis, we design a novel DSL optimized for sequential decision-making, structuring actions into three composable sub-actions: color selection, cell selection, and transformations. This design enables policy learning by decomposing complex tasks into interpretable steps. We implement a custom RL environment where agents interact with ARC tasks by iteratively applying DSL operations. To address the challenge of large discrete action spaces, we integrate the Wolpertinger architecture, combining Deep Deterministic Policy Gradient (DDPG) with k-nearest neighbors (KNN) to map continuous proto-actions to discrete DSL commands. State representations are encoded via a convolutional neural network (CNN) augmented with self-attention, enhancing spatial reasoning. Experimental results demonstrate that our framework learns structured transformations, evidenced by monotonically decreasing policy and value loss. However, generalization to unseen ARC tasks remains limited under current computational constraints. We identify hierarchical policy decomposition and learnable action embeddings as critical directions for improving adaptability and scalability.

**Keywords:** Reinforcement Learning (RL), Abstraction and Reasoning Corpus (ARC), Domain-Specific Language (DSL), Deep Deterministic Policy Gradient (DDPG), Wolpertinger Architecture

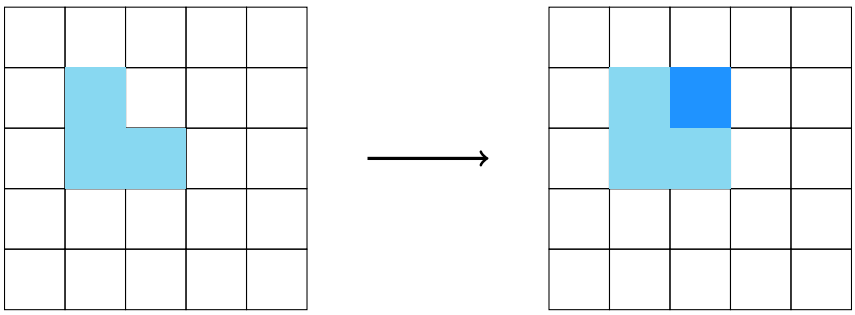
Code available at: https://github.com/francescobraicovich/ARC.git

# Contents

# 1 Introduction

## 1.1 Abstract reasoning corpus

The Abstraction and Reasoning Corpus (ARC), introduced in the seminal work *On the Measure of Intelligence* [1], serves as a benchmark for artificial general intelligence (AGI) designed to evaluate an AI system's ability to acquire new skills and solve novel, abstract problems through few-shot learning and broad generalization. The corpus comprises 1,000 tasks, with 800 publicly accessible (divided equally into 400 training and 400 evaluation tasks) and 200 reserved for a hidden test set. Each task is defined by a small set of input-output examples, where the input grid is transformed into the output grid via a task-specific rule. The core challenge lies in inferring this underlying rule from few (maximum 4) demonstrations and applying it accurately to unseen inputs.



Input Grid Target Grid

ARC tasks operate within a constrained domain: grids are limited to a resolution of $30 \times 30$ pixels, each assigned one of 10 distinct colors. Despite these simplifications, the tasks span a diverse array of concepts, including object recognition, arithmetic operations, geometric transformations (e.g., mirroring, rotation), pattern completion, and conditional object manipulation (e.g., movement, alteration, or removal). While human adults solve approximately 80% of tasks through intuitive reasoning, state-of-the-art AI systems struggle significantly. This disparity stems from ARC's emphasis on rapid skill acquisition and systematic generalization—capabilities not inherently supported by conventional machine learning paradigms, which often rely on extensive data and predefined objectives.

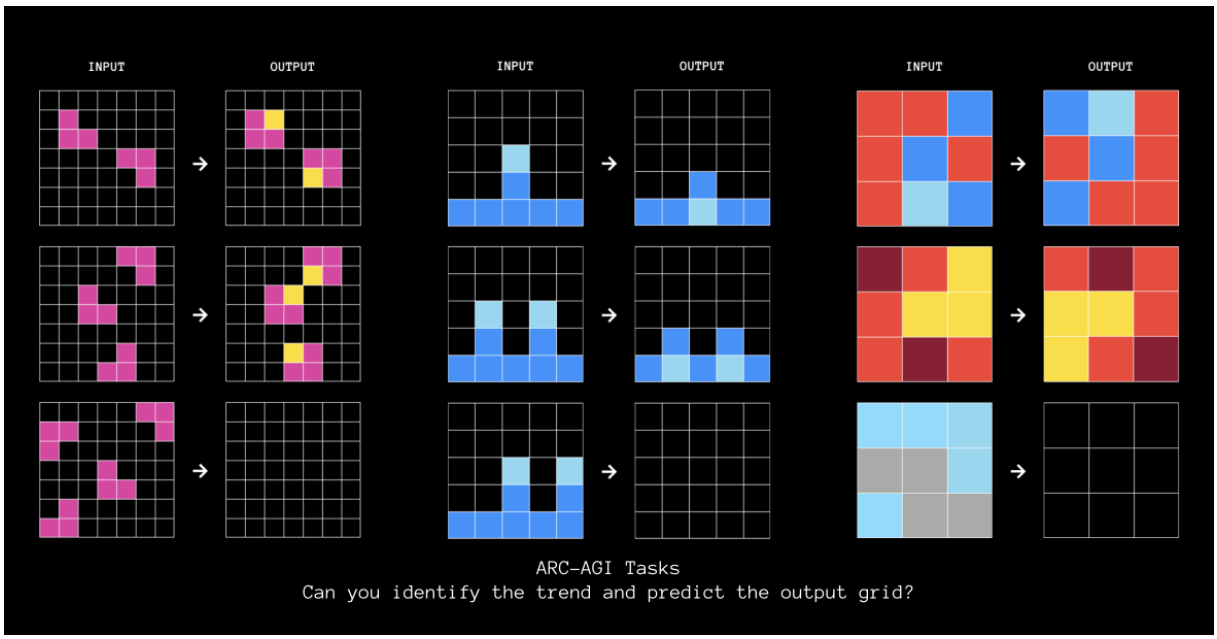



**Figure 1.1:** Real ARC Examples from https://arcprize.org/blog/launch

Existing computational approaches to ARC broadly fall into two categories:

- **Induction-Based Methods** These aim to construct a programmatic function $f$ that maps training inputs $x_{\text{train}}$ to outputs $y_{\text{train}}$ via interpretable transformations (e.g., rotations, reflections, color operations). The derived function is then applied to test inputs $x_{\text{test}}$ .
- **Transduction-Based Methods:** In contrast, transductive approaches bypass explicit rule extraction, treating the model itself as a dynamic program that directly generates outputs for novel inputs.

Both paradigms face significant hurdles in scaling to ARC's complexity, underscoring the benchmark's utility in measuring progress toward more adaptable, human-like reasoning in AI systems.

## 1.2 Proposed Approach

We propose a framework that integrates **reinforcement learning (RL) for sequential transformation prediction**, inspired by human cognitive learning. Empirical studies in cognitive science [2, 3] suggest that humans solve novel tasks through iterative hypothesis testing, refining strategies based on observed outcomes. This trial-and-error process aligns with model-free RL, where an agent explores a sequence of transformations, receives feedback, and updates its policy accordingly.
Our approach **trains a model to sequentially predict the next best transformation**, given the current grid and target output grid. The RL agent iterates over transformation sequences, selecting primitives that maximize alignment with the desired output. During training, the agent learns policies that generalize across tasks by predicting one transformation at a time. When tested, instead of relying solely on inference, we employ **transfer learning to fine-tune the model on the specific task**, allowing it to overfit on available examples. Due to the structured nature of our **Domain-Specific Language (DSL)**, a transformation sequence that successfully maps one example within a task should generalize to the remaining examples. This approach ensures adaptability while leveraging prior knowledge efficiently.

### 1.2.1 Methodology

Our framework consists of two core components:

1. **Domain-Specific Language (DSL):** We introduce a structured language designed for sequential transformation synthesis, incorporating primitives such as rotation, mirroring, filtering, and object detection. While robust DSLs already exist [4], our DSL is tailored for RL-based learning, enabling stepwise transformation selection.
2. **Sequential Program Search with RL:** We formulate program synthesis as a sequential decision-making problem, where an RL agent predicts the next best transformation at each step. To efficiently navigate the large action space, we employ the Wolpertinger architecture, an actor-critic framework optimized for discrete decision spaces. While transformers are well-suited for encoding long-range dependencies, their computational cost is prohibitive. Instead, we utilize convolutional neural networks (CNNs) for both the actor and critic networks, with a self attention layer.

# 2 Domain-Specific Language (DSL)

## 2.1 Design Principles and Scope

To effectively represent and manipulate ARC tasks, we introduce a domain-specific language (DSL) tailored for efficient reasoning over transformation sequences. The DSL defines a set of operations—referred to as *actions*—that include geometric transformations (e.g., rotations, reflections, scaling), color modifications, and pattern recognition. A well-designed DSL is essential for achieving sample efficiency, as ARC tasks typically provide only two solved examples per problem. By structuring transformations within a compact and expressive DSL, we ensure that if a sequence of actions correctly maps the input to the output in the given examples, it generalizes to new instances by design.
Our DSL satisfies the following key principles:

1. **Expressiveness:** The DSL must be capable of representing a broad range of ARC tasks through its primitives.
2. **Abstraction and Generality:** The DSL defines a minimal set of core primitives, each applicable across multiple tasks, ensuring efficiency without sacrificing versatility.
3. **Sequentiality:** Since our approach models program synthesis as a sequential decision-making process, the DSL must facilitate stepwise transformation application, where each transformation modifies the grid based only on its current state.

The DSL proposed by Hodel [5] is designed around the principles of expressiveness and abstraction, ensuring that a broad range of ARC tasks can be represented with a minimal yet powerful set of primitives. While our DSL retains these foundational characteristics, it diverges fundamentally in its sequential nature. In contrast to Hodel's formulation, where transformations can be expressed as a holistic function over the entire input-output mapping, our DSL enforces an iterative application of transformations, each conditioned solely on the current state of the grid. This sequential structure is essential for integrating reinforcement learning (RL), where an agent selects and applies transformations dynamically based on observed states, without explicit access to prior states.

### 2.1.1 Action Decomposition

Each action in our DSL is decomposed into three fundamental sub-actions:

1. **Color Selection:** Identifies relevant colors within the grid.
2. **Cell Selection:** Determines which cells, given the selected color, will be modified.
3. **Transformation:** Applies a geometric or structural transformation to the selected cells.

This structured decomposition ensures that transformations are both interpretable and modular, facilitating effective policy learning within the RL framework. A full specification of the DSL, including its formal syntax and semantics, is provided in Appendix A.

## 2.2 Color Selection

The first part of the DSL, that is the one devoted to the color selection, provides a set of methods to choose a color from a grid according to various possible rules. The chosen color is used as input of the selection or transformation methods that need one, in order to construct a complete action. These rules should be simple and capture patterns that an average person can recognize with minimal effort. The goal is to ensure that the patterns align with intuitive human perception, as this is one of the core goals of the challenge. The principal methods consist of selecting the color according to its appearance rank in the whole grid or in some specific shapes. They require as input a 2D grid (and sometimes a rank) and output an integer value between 0 and 9, that corresponds to a specific color. Table 2.1 summarizes the main color selection functions.

| Function(s) | Description |
|---|---|
| `mostcolor`, `leastcolor` | Flattens the grid and counts color frequencies. Returns the most common color with `mostcolor` or, after replacing zero counts with a large number, the least common color with `leastcolor`. |
| `rankcolor` | Sorts unique colors by frequency and returns the color at the specified rank (defaults to the least common if the rank is too high). |
| `rank_largest_shape_color_nodiag` | Labels non-diagonally connected regions for each color and returns the color corresponding to the rank-th largest connected shape. |

**Table 2.1:** Summary of Color Selection Functions.

## 2.3 Selection

The following step in building an action consists in choosing the object of the transformation. This amounts to providing a tool to select a part of the grid based on different possible strategies, that should be again be simple and recognizable by an average human being. These methods offer the possibility to select cells matching a target color, extract particular kinds of rectangular or square regions, identify connected shapes using either 4 or 8-connectivity, and delineate both inner and outer borders of these regions. Methods that require a specific color have both the possibility to use colors that are output of a color selection and colors that are not, since different tasks may require both possibilities.
Some selection methods can return multiple results; for example, there may be several rectangles of a given color within the grid. To accommodate this, each selection method takes a 2D grid as input but produces a 3D output, where each layer is a separate 2D boolean mask. These masks have 1s indicating the selected cells and 0s elsewhere, allowing to stack multiple selections into a single output structure. An example of how this works can be found in Figure 2.1. Table 2.2 summarizes the main selection functions.

| Function(s) | Description |
|---|---|
| `select_color` | Creates a boolean mask selecting all cells of a specified color. |
| `select_rectangles` | Selects rectangular regions of a given height and width that are entirely filled with the target color. |
| `select_connected_shapes`, `select_connected_shapes_diag` | Identifies and extracts connected shapes of a specified color. The first function uses 4-connectivity, while the second includes diagonal connections (8-connectivity). |
| `select_adjacent_to_color`, `select_adjacent_to_color_diag` | Selects cells adjacent to a target color. The first considers only edge-connected neighbors (4-connectivity), while the second includes diagonally adjacent cells (8-connectivity). |
| `select_outer_border`, `select_inner_border` | Extracts the outer or inner borders of edge-connected shapes of a given color. |
| `select_outer_border_diag`, `select_inner_border_diag` | Extracts the outer or inner borders of shapes using 8-connectivity, including diagonal neighbors. |
| `select_all_grid` | Creates a selection covering the entire grid. |

**Table 2.2:** Summary of DSL Selection Functions.

## 2.4 Transformation

Lastly we introduce the main transformation functions that our agent should be able to operate. In our implementation, these functions (40) are organized as methods of a `Transformer` class. They enable a relatively wide range of operations including color manipulation, flipping, rotation, cropping, copy-paste operations, gravity-based movements, and upscaling. These functions are designed to work on a 2D grid (represented as a NumPy array) and a 3D boolean mask that indicates the selection; the grid is stacked along a third dimension using the `create_grid3d` utility function. Table 2.3 summarizes the main transformation functions.

| Function(s) | Description |
|---|---|
| `flipv`, `fliph`, `flip_main_diagonal`, `flip_anti_diagonal`,`mirror_up/down` `mirror_left/right`,`duplicate` | Perform various flipping operations: vertical/horizontal flips, mirroring along main or anti-diagonals, and duplicating the selection in specified directions (subject to grid constraints). |
| `rotate90`, `rotate180`, `rotate270` | Rotate the selected cells by 90°, 180°, or 270° counterclockwise. |
| `new_color`, `color`, `fill_with_color`, `fill_with_color`, `change_color`, | Apply various color transformations: change cell color, fill shapes with a specified color, and adjust background or selection colors using ranking or shape criteria. |
| `copy_paste`, `copy_sum`, `cut_paste`, `cut_sum`, `copy_paste_vertically`, `copy_paste_horizontally` | Execute copy-paste operations (or cut-paste) with or without summing values, including repeated pasting vertically or horizontally within grid bounds. |
| `gravitate_whole_down/upwards_paste` `gravitate_whole_right/left_paste` | Copy-paste the entire selection in a specified direction until an obstacle or grid edge is encountered. |
| `gravitate_whole_down/upwards_cut`, `gravitate_whole_right/left_cut`, | Cut-paste the entire selection in the specified direction until an obstacle or grid boundary is reached. |
| `down_gravity`, `up_gravity`, `right_gravity`, `left_gravity` | Apply gravity to individual selected cells, moving them in the specified direction until they hit an obstacle or the grid boundary. |
| `vupscale`, `hupscale`, `vectorized_vupscale` | Upscale the selection vertically or horizontally by a given factor while capping to the original grid size; includes a vectorized vertical upscaling variant. |
| `crop`, `delete` | Crop the grid to the bounding rectangle of the selection (cells outside are set to -1) or set the selected cells to zero. |

**Table 2.3:** Summary of DSL Transformation Functions.

In the Appendix A a detailed overview of the transformation functions is present. This framework forms the backbone of our approach and is the essential set of actions through which our agent operates. Thanks to its modularity we allow for flexible exploration of transformation operations, which is critical for addressing the diverse and few-shot nature of ARC tasks.

## 2.5 A Concrete Example

In the following, we illustrate a complete action. The first grid represents the current state, where *mostcolor* selects Blue. This color is then used by *Select_connected_shapes* to create a 3D grid—a stack of two 2D black-and-white masks (black = 1, white = 0). Next, the transformation *rotate90* is applied, producing a 3D grid of the same dimensions. Note that a rotation works on squares, hence when the selection does not output a square, the rotation method applies the transformation on the smallest bounding square that contains the original selection. Highlighted in orange are showed the bounding squares.

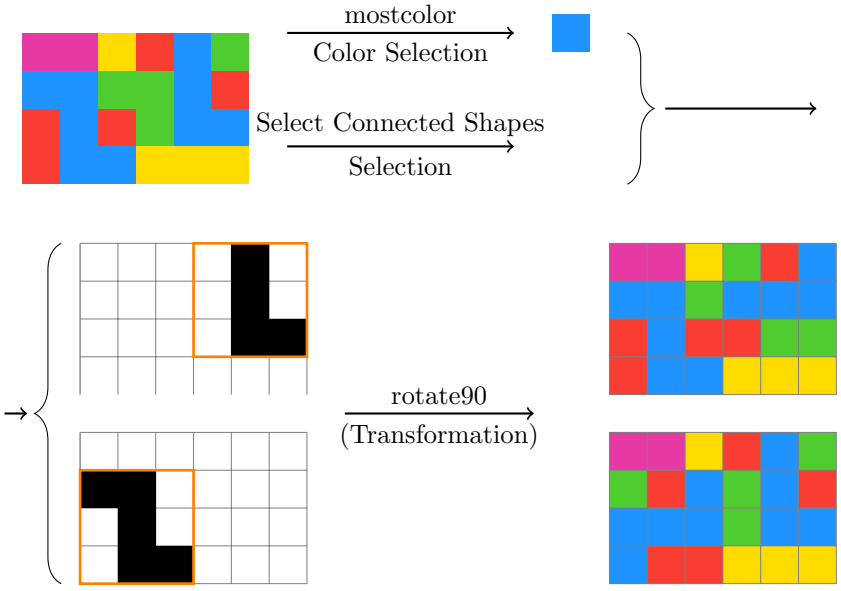



**Figure 2.1:** Complete process showing color selection, shape identification, and grid transformation

# 3 Action Space & Environment

## 3.1 Action Space

As outlined in the previous section, the Domain-Specific Language (DSL) consists of three categories of functions: **color selection**, **selection**, and **transformation**. We define the *action space* $\mathcal{A}$ as the set of all possible triplets formed by selecting one function from each of these categories:

$$\mathcal{A} := \left\{ a = (\text{color\_selection}_i, \text{selection}_j, \text{transformation}_k) \;\middle|\; i = 1, \ldots, n_{cs};\ j = 1, \ldots, n_s;\ k = 1, \ldots, n_t \right\},$$

where $n_{cs}$, $n_s$, and $n_t$ denote the total number of color selection, selection, and transformation functions, respectively. Each function within a category is assigned a value in the interval $[0, 1]$, resulting in the action space $\mathcal{A}$ being a subset of $\mathbb{R}^3$ by construction. Our DSL currently allows for approximately 20000 actions.

### 3.1.1 Action Similarity

The Wolpertinger architecture (see §4.3) leverages K-Nearest Neighbors to identify the closest actions in a discrete space, given a proto-action (the output of the actor network) in a continuous space. For this approach to be effective, it is crucial that similar actions in the discrete space are mapped close to each other in the continuous space. To achieve this, we construct a new space $\tilde{\mathcal{A}}$, referred to as the *Space of Embedded Actions*. This space is built by computing the following three similarity matrices:

1. **Color Similarity Matrix** $C \in \mathbb{R}^{n_{cs} \times n_{cs}}$: This matrix quantifies the similarity between pairs of color selection functions by estimating the probability that two functions select the same color when applied to random inputs. Given $n_e$ random inputs $s_i$, the matrix is defined as:

$$C = \left[ \frac{1}{n_e} \sum_{i=1}^{n_e} \mathbb{1} \left( \text{color\_selection}_j(s_i) = \text{color\_selection}_k(s_i) \right) \right]_{\substack{j=1,\ldots,n_{cs} \\ k=1,\ldots,n_{cs}}}$$

2. **Selection Similarity Matrix** $S \in \mathbb{R}^{n_s \times n_s}$: This matrix measures the similarity between pairs of selection functions based on the average overlap of their outputs when applied to the same random input and color selection. Given $n_e$ random inputs $s_i$ and random colors $c_i$, the matrix is defined as:

$$S = \left[ \frac{1}{n_e} \sum_{i=1}^{n_e} \text{MaxOverlap} \left( \text{selection}_j(s_i, c_i), \text{selection}_k(s_i, c_i) \right) \right]_{\substack{j=1,\ldots,n_s \\ k=1,\ldots,n_s}}$$

   where $\text{MaxOverlap}(\cdot, \cdot)$ is formally defined in §3.2.2.

3. **Transformation Similarity Matrix** $T \in \mathbb{R}^{n_t \times n_t}$: This matrix quantifies the similarity between pairs of transformation functions by evaluating the average overlap of their outputs when applied to the same random input grid and selection. Given $n_e$ random inputs $s_i$ and selections $\text{sel}_i$, the matrix is defined as:

$$T = \left[ \frac{1}{n_e} \sum_{i=1}^{n_e} \text{MaxOverlap} \left( \text{transformation}_j(s_i, \text{sel}_i), \text{transformation}_k(s_i, \text{sel}_i) \right) \right]_{\substack{j=1,\ldots,n_t \\ k=1,\ldots,n_t}}$$

**Action Pair Similarity:** Using these matrices, we define the approximate similarity between two actions $a_i = (\text{color\_selection}_p, \text{selection}_q, \text{transformation}_r)$ and $a_j = (\text{color\_selection}_u, \text{selection}_v, \text{transformation}_w)$ as the product of the similarities of their respective components:

$$\text{Sim}(a_i, a_j) = C_{p,u} \cdot S_{q,v} \cdot T_{r,w}$$

This approximation allows for efficient computation of action similarities in this highly combinatorial spaces.

### 3.1.2 Action Embedding with MDS

Given the similarity between all pairs of actions, we define the distance between actions $a_i$ and $a_j$ as:

$$d(a_i, a_j) = 1 - \text{Sim}(a_i, a_j),$$

where $\text{Sim}(a_i, a_j)$ represents the similarity between actions as previously defined. This transformation ensures that highly similar actions have smaller distances, while dissimilar actions have larger distances. To embed the discrete set of actions $\mathcal{A}$ into a continuous space, we employ the *Multidimensional Scaling* (MDS) algorithm. MDS aims to represent high-dimensional data in a lower-dimensional space such that the pairwise distances between points are preserved as much as possible.

**Formal Definition of MDS:** Let $D = [d(a_i, a_j)]$ be the distance matrix of all action pairs, where $D \in \mathbb{R}^{|\mathcal{A}| \times |\mathcal{A}|}$. The goal of MDS is to find a set of vectors $\{\tilde{a}_i\}_{i=1}^{|\mathcal{A}|}$ in $\mathbb{R}^d$ such that the Euclidean distances between the embedded actions $\tilde{a}_i$ and $\tilde{a}_j$ approximate the original distances $d(a_i, a_j)$:

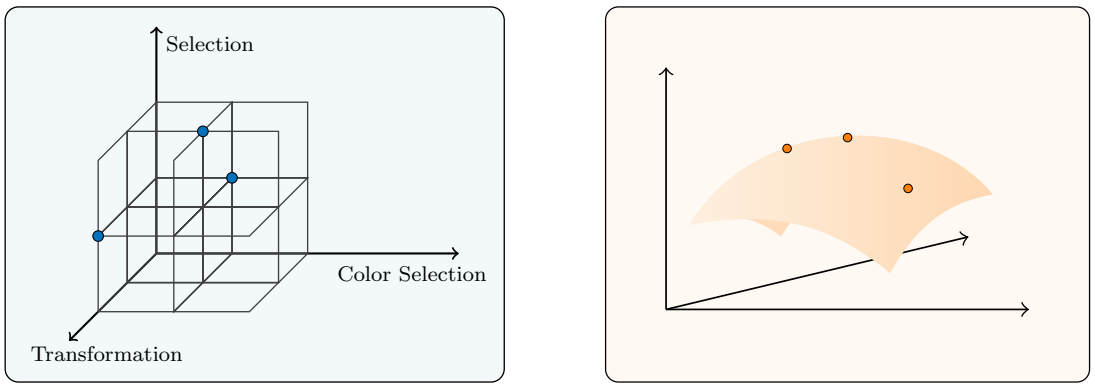
$$\|\tilde{a}_i - \tilde{a}_j\|_2 \approx d(a_i, a_j), \quad \forall i, j.$$

This is achieved by minimizing the following stress function:

$$\text{Stress}(\{\tilde{a}_i\}) = \sqrt{\sum_{i<j} \left(d(a_i, a_j) - \|\tilde{a}_i - \tilde{a}_j\|_2\right)^2}.$$

In our case, we embed the actions into a 20-dimensional continuous space $\tilde{\mathcal{A}} \subset \mathbb{R}^{20}$, where Euclidean distances approximate the original distances derived from the similarity measures.

**Benefits of MDS Embedding:** This embedding provides a continuous space $\tilde{\mathcal{A}}$ where each action $a \in \mathcal{A}$ has a corresponding embedding $\tilde{a} \in \tilde{\mathcal{A}}$. Actions that are similar in their discrete representations are mapped close to each other in this continuous space. The proto-action output from the actor network resides in $\tilde{\mathcal{A}}$, and the K-Nearest Neighbors (KNN) algorithm (see §4.3) is used to identify the closest discrete actions in $\mathcal{A}$. This ensures that the actor network can efficiently navigate the action space, leveraging the proximity of similar actions in the embedded space.




Original Discrete Space $\mathcal{A} \in \mathbb{R}^3$ | Embedded Continuous Space $\tilde{\mathcal{A}} \in \mathbb{R}^d$

**Figure 3.1:** Visualization of the action embedding process. The left panel illustrates the original discrete action space $\mathcal{A} \in \mathbb{R}^3$, defined by the combination of color selection, selection, and transformation functions. The right panel shows the embedded continuous action space $\tilde{\mathcal{A}} \in \mathbb{R}^d$ (with $d > 3$), obtained through Multidimensional Scaling (MDS), where similar actions from the discrete space are mapped to nearby points in the continuous space.

## 3.2 Environment

We introduce `ARC_Env`, an environment designed to formulate Abstraction and Reasoning Corpus (ARC) tasks as a reinforcement learning (RL) problem within the Gymnasium framework.

### 3.2.1 Markov Decision Process Formulation.

The environment is structured as a Markov Decision Process (MDP) $(S, A, R, \gamma)$, where:

- $S$ is the set of states, each representing a 2D grid padded to a uniform size of $30 \times 30$, corresponding to the maximum grid size in ARC.
- $A$ is the set of actions, each composed of a color selection, a selection operation, and a transformation.
- $R(s, a)$ is the reward function, measuring the improvement in alignment between the transformed grid and the target.
- $\gamma$ is the discount factor, regulating the importance of future rewards.

At each timestep $t$, the agent observes the current state $s_t$, selects an action $a_t \in A$, receives a reward $R(s_t, a_t)$, and transitions to a new state $s_{t+1}$. The episode terminates if the agent reconstructs the target grid, the grid degenerates to a single color,[1] or a maximum of 30 steps is reached.
Each state consists of:

- **Current Grid:** The agent's current representation of the task.
- **Target Grid:** The goal configuration that the agent must reconstruct.

All grids are padded to a fixed $30 \times 30$ dimension before being served to the network. However, transformations and reward computations operate on the unpadded version of the grids to ensure meaningful comparisons.
Each transformation produces a set of possible outputs, resulting in a 3D grid where each layer represents a candidate transformation.

### 3.2.2 Reward Function

The reward function evaluates how much a transformation improves the alignment between the transformed grid and the target. It is defined as follows:

$$R(s, a) = \max_{\Delta x, \Delta y} \sum_{i,j} \mathbb{1}[G'_{i+\Delta x, j+\Delta y} = T_{i,j}] - \max_{\Delta x, \Delta y} \sum_{i,j} \mathbb{1}[G_{i+\Delta x, j+\Delta y} = T_{i,j}]. \tag{3.1}$$

Here,

- $G$ is the current grid before the transformation.
- $G'$ is the transformed grid after applying action $a$.
- $T$ is the target grid.
- $\mathbb{1}[P]$ is the indicator function, which evaluates to 1 if the predicate $P$ is true and 0 otherwise.
- The max operator computes the highest possible overlap between the transformed grid $G'$ and the target $T$, allowing for translations in both horizontal ($\Delta x$) and vertical ($\Delta y$) directions.

This function computes two quantities:

1. The maximum overlap between the transformed grid $G'$ and the target grid $T$ over all possible translations.
2. The maximum overlap between the original grid $G$ and the target grid $T$ before the transformation.

[1] If the grid collapses to a uniform color, the episode is truncated because such a state is typically irreversible, making it nearly impossible for the agent to recover meaningful progress. Assigning a large negative reward discourages actions that lead to this failure mode.

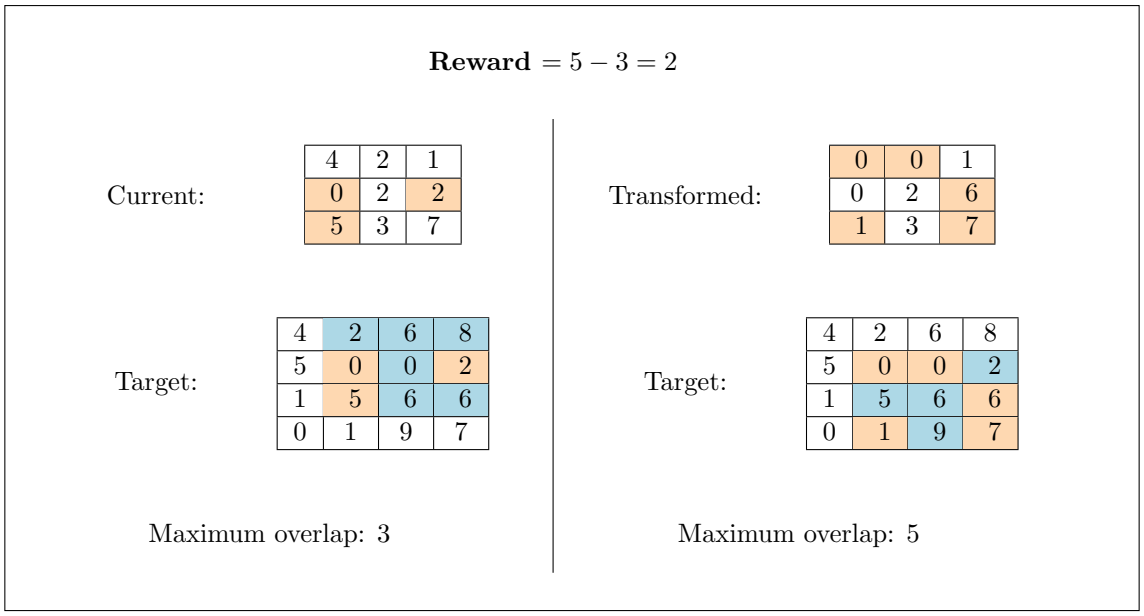



**Figure 3.2:** Example of calculation of the reward function.

By computing the difference between the post-transformation and pre-transformation maximum overlaps, the reward function captures whether an action moved the agent closer to solving the task. If the transformation reduces overlap (i.e., worsens alignment), the reward is negative, penalizing ineffective actions.
Since each transformation generates multiple possible outputs, the environment initially represents the transformed state as a 3D grid where each layer corresponds to a different possible transformation result. However, during reward computation, only the transformed state with the highest overlap with the target is selected. This ensures that the representation remains in 2D after the reward function is applied.

# 4 Architecture

Large discrete action spaces pose significant challenges to conventional Reinforcement Learning (RL) algorithms that typically assume either:

- Small discrete sets (e.g., Atari-like settings), or
- Continuous controls (e.g., MuJoCo environments).

Examples of large discrete tasks include ARC-like puzzles, where a Domain-Specific Language (DSL) might combine color operations, shape transformations, and other primitives to form tens of thousands (or more) possible actions.
To address this, we combine:

1. **DDPG** [6], an off-policy actor-critic approach typically geared towards continuous action spaces.
2. **Wolpertinger** [7], which adapts DDPG to large discrete action sets by mapping a *continuous proto-action* to a discrete action via $\mathcal{N}_k(\hat{a})$, the set of $k$-nearest neighbors.
3. **Feature Extraction for the Actor and Critic Networks**:
   - **Latent Program Network (LPN) Encoder** [8], leveraging a Bayesian framework to produce latent representations of partial program/task specifications. We omit the decoder and retain only the encoder side.
   - **CNN + Multi-Head Self-Attention**, a modified convolutional backbone (inspired by ResNet) plus an attention module to capture broad context from grid-based states.

In the following sections, we delve into the technical details behind each component. We start with an explanation of how the actor and critic function within DDPG, then discuss Wolpertinger's discrete selection strategy, expand on the Bayesian viewpoint in the LPN encoder, and finally describe our CNN with *multi-head* self-attention. We conclude with the training loop. Schematic diagrams (Figures 4.1-4.4) illustrate the flow of data and computations.

## 4.1 High-Level Architecture Overview

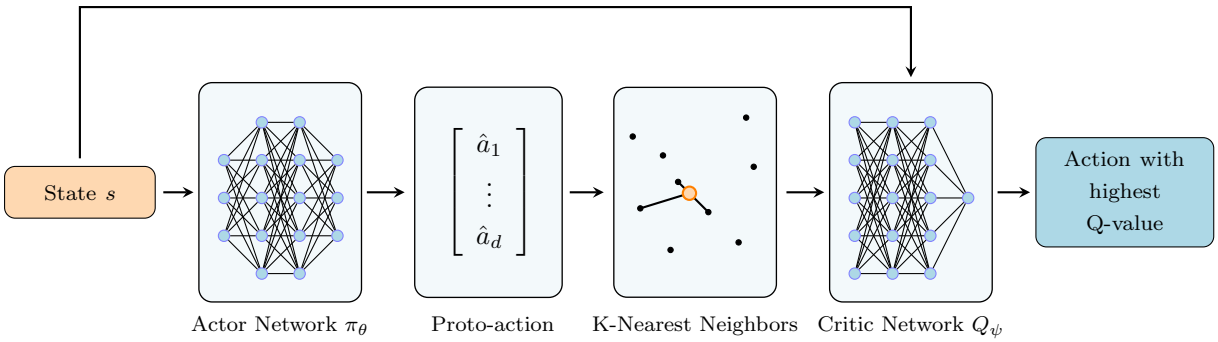



**Figure 4.1:** Overall Architecture: The state $s$ is processed by the actor network $\pi_\theta$, which generates a continuous proto-action $\hat{a} \in \tilde{\mathcal{A}} \subset \mathbb{R}^d$. The K-Nearest Neighbors algorithm identifies the set of actions $\mathcal{N}k(\hat{a}) \in \tilde{\mathcal{A}}$ that are closest to $\hat{a}$. The critic network $Q_\psi$ evaluates these candidate actions by taking both the state $s$ and the nearest actions $\mathcal{N}_k(\hat{a})$ as inputs to estimate their corresponding $Q$-values. The final action selected is the one whose corresponding embedded action has the highest $Q$-value.

## 4.2 Deep Deterministic Policy Gradient (DDPG)

DDPG is an off-policy RL algorithm that extends deterministic policy gradients to high-dimensional continuous spaces. Here, the idea is adapted: the output of the actor is still a continuous vector (proto-action), but we will use Wolpertinger to select discrete actions. In our implementation, the state space $\mathbb{S}$ is composed of elements:

$$s \in \mathbb{S} = (grid_t, grid_{t+1})$$

where:

- $t \geq 0$ is a given timestep.
- $grid_t$ is the grid up to the previous action.
- $grid_{t+1}$ is the grid after the action at time $t$.

Both grids are padded to match the maximum grid size of $(30 \times 30)$

### 4.2.1 Critic Network

**Definition.** The critic, denoted as $Q_\psi : \mathcal{S} \times \tilde{\mathcal{A}} \to \mathbb{R}$, estimates the $Q$-value for a given state-action pair $(s, \tilde{a})$. The critic receives actions in their embedded form, as detailed in §3.1.2. Formally, the critic is defined as:

$$Q_\psi(s, \tilde{a}) = \underbrace{\text{MLP}_\psi}_{\text{Final Layers}} \Big( \underbrace{\phi_\psi(s)}_{\text{Encoded State}}, \tilde{a} \Big).$$

Where:

- The critic network is parametrised by $\psi$.
- $\phi_\psi(s)$ is the encoded state (§4.4) as parametrised by $\psi$.
- $\tilde{a} \in \tilde{\mathcal{A}} \subset \mathbb{R}^d$ is the embedding of an action $a \in \mathcal{A}$, the discrete action space.

The encoded state $\phi_\psi(s)$ and the embedded action $\tilde{a}$ are concatenated into a single vector $\big[\phi_\psi(s), \tilde{a}\big] \in \mathbb{R}^{d_s+d}$. This concatenated vector is then passed through a feed-forward network (MLP) parameterized by $\psi$, producing a scalar $Q$-value (see Fig. 4.3).

**Objective.** We train the critic by minimizing the following loss function:

$$L_{\text{critic}}(\psi) = \frac{1}{B} \sum_{i=1}^{B} \Big( \big[ r_i(s_i, a_i) + \gamma y_i \big] - Q_\psi(s_i, \tilde{a}_i) \Big)^2,$$

Where:

- $r(s, a)$ represents the reward provided by the environment after performing the action $a$ in state $s$. It's important to note that the environment processes the non-embedded version of the action.
- $y_i$ is the predicted $Q$-value of the next state $s'$, obtained using the Target Critic Network. The corresponding next action $\tilde{a}$ is computed using the Target Actor Network (refer to Section §4.2.3).

where $y_i$ are target $Q$-values computed via target networks and Wolpertinger (see §4.3).

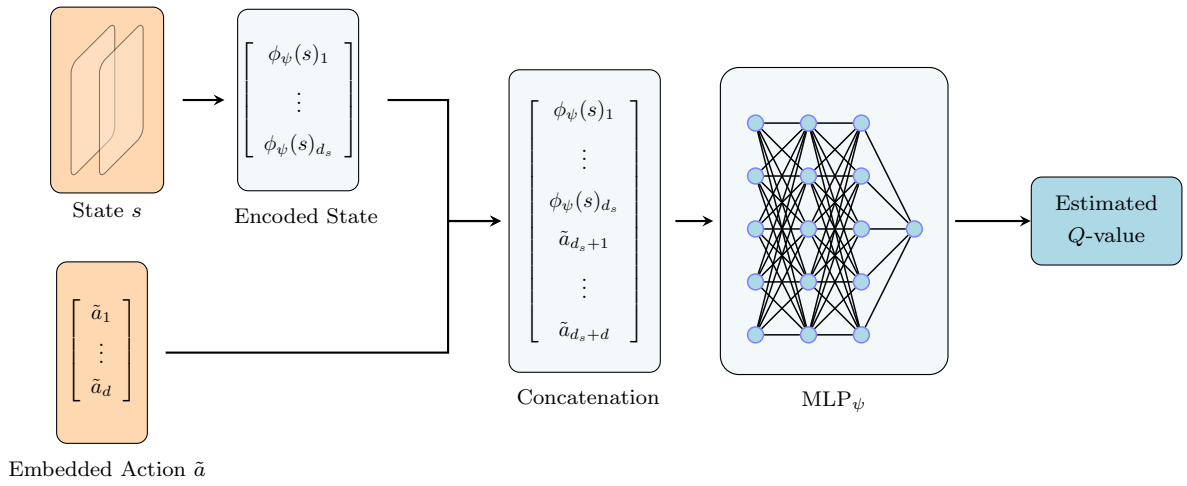



**Figure 4.2:** Critic structure: The critic concatenates the state embedding $\phi(s)$ and the action embedding $\psi_a(a)$, then processes the result with a feed-forward network to yield a scalar Q-value.

### 4.2.2 Actor Network

**Definition.** The actor, denoted as $\pi_\theta : \mathcal{S} \to \mathbb{R}^d$, maps each state $s \in \mathcal{S}$ to a $d$-dimensional *proto-action* $\hat{a}$. Formally, the actor is defined as:

$$\pi_\theta(s) = \underbrace{\text{MLP}_\theta}_{\text{Final Layers}} \Big( \underbrace{\phi_\theta(s)}_{\text{Encoded State}} \Big).$$

Where:

- The actor network is parametrized by $\theta$.
- $\phi_\theta(s)$ is the encoded state (§4.4) as parametrized by $\psi$.

The encoded state $\phi_\theta(s)$ is passed through a feed-forward network (MLP) parameterized by $\theta$, producing a proto-action $\hat{a}$.

**Objective.** We train the actor to maximize the estimated $Q$-values computed by critic (minimize the additive inverse of the estimated $Q$-values). The loss function for the actor is defined as:

$$L_{\text{actor}}(\theta) = -\frac{1}{B} \sum_{i=1}^{B} Q_\psi\Big( s_i,\ \pi_\theta(s_i) \Big),$$

Where $Q_\psi$ is the critic network that evaluates the quality of the proto-action $\hat{a}$ in the given state $s$. The actor is optimized to increase the critic's estimated $Q$-values by adjusting the parameters $\theta$, thereby improving the policy over time.

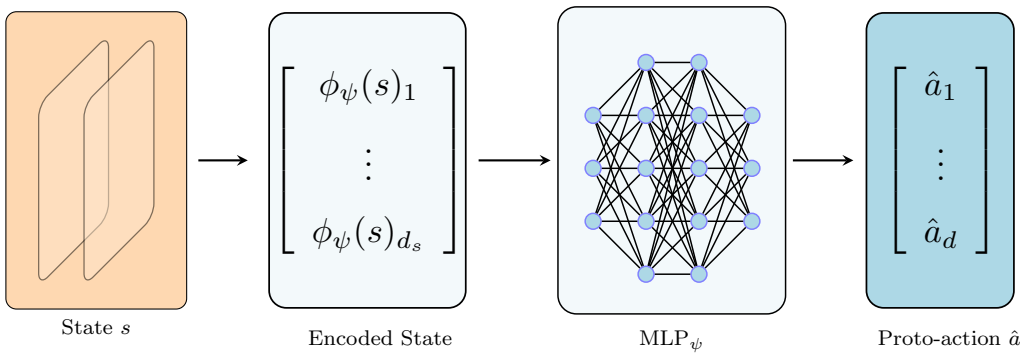



**Figure 4.3:** Actor structure: The actor takes as input state $s$ and outputs a proto-action $\hat{a}$

### 4.2.3 Target Networks and Replay Buffer

- **Target Networks:** $(\theta_{\text{target}}, \psi_{\text{target}})$ are slowly updated copies of $(\theta, \psi)$ to stabilize Q-value targets. After each mini-batch update, we do:

$$\theta_{\text{target}} \leftarrow \tau\,\theta + (1-\tau)\,\theta_{\text{target}}, \quad \psi_{\text{target}} \leftarrow \tau\,\psi + (1-\tau)\,\psi_{\text{target}},$$

  with a small $\tau \in (0, 1]$.

- **Replay Buffer:** We store transitions $(s, a, r, s', \text{done})$. During each training iteration, we sample a batch of transitions to compute $L_{\text{critic}}$ and $L_{\text{actor}}$.

## 4.3 Wolpertinger for Large Discrete Actions

In classical DDPG, $\pi_\theta(s) \in \mathbb{R}^d$ directly outputs the continuous action. Wolpertinger [7] adapts this architecture to large discrete action spaces, by leveraging K-Nearest Neighbors. Given the proto-action $\hat{a} = \pi_\theta(s)$, we identify the $k$ nearest embedded actions in the discrete space $\tilde{\mathcal{A}}$:

$$\mathcal{N}_k(\hat{a}) = \arg\min_{\tilde{a} \in \tilde{\mathcal{A}}} \|\hat{a} - \tilde{a}\|^2 \quad (\text{top } k).$$

Among these $k$ nearest neighbors, we select:

$$\tilde{a}^* = \arg \max_{\tilde{a} \in \mathcal{N}_k(\hat{a})} Q_\psi(s, \tilde{a}).$$

This is the embedded action among those in the K-Nearest Neighbors, with the highest estimated $Q$-value. The corresponding action $a \in \mathcal{A}$ will be performed on state $s$. Hence, the actor indirectly selects a discrete action by positioning its continuous proto-action $\hat{a}$ near the desired region in $\mathbb{R}^d$.

**Complexity.** Exact $k$-NN lookups can have a computational complexity of $O(|\mathcal{A}|\, d)$ at each step. If $|\mathcal{A}|$ is very large, approximate methods (e.g., KD-trees, hierarchical embeddings) are recommended. Gradient flow from the Q-function to the actor bypasses $\mathcal{N}_k(\hat{a})$ because the discrete selection is non-differentiable; the actor learns to place $\hat{a}$ in favorable neighborhoods over time.

## 4.4 Encoding States

### 4.4.1 Latent Program Network (LPN) Encoder with Bayesian Framework

Bonnet and Macfarlane [8] propose an encoder-decoder approach for program synthesis, incorporating a *Bayesian* perspective:

- **Encoder:** $q_\theta(z \mid x)$ produces a *posterior* distribution over latent codes $z \in \mathbb{R}^D$ given partial program specifications $x$ (e.g., example input-output pairs, constraints, partial code).
- **Decoder:** $p_\phi(x \mid z)$ reconstructs or samples a DSL program from the latent code $z$.
- **Variational Objective:** The following Evidence Lower Bound (ELBO) is optimized:

  $$\mathcal{L}(\theta, \phi) = \mathbb{E}_{q_\theta(z|x)}\big[\log p_\phi(x \mid z)\big] \; - \; \beta\, \mathrm{KL}\big(q_\theta(z \mid x) \,\|\, p(z)\big),$$

  where $p(z)$ is typically a standard Gaussian prior. This encourages $z$ to encode meaningful variations of $x$ while maintaining a smooth, expressive latent space.

**How We Use It.** Instead of decoding full programs, we:

1. **Retain only the encoder**, i.e., $z = \mathrm{Enc}_\theta(s)$.
2. Optionally treat $z$ as a random sample from $q_\theta(z \mid s)$; however, in practice, we often use the mean $\mu_\theta(s)$ for stability.
3. This $z \in \mathbb{R}^D$ is then passed to an MLP that outputs $\hat{a} \in \mathbb{R}^d$. Thus, $\pi_\theta(s)$ is effectively a composition of LPN Encoder and MLP.

Because the encoder was trained (or co-trained) under a Bayesian framework, $z$ can reflect a distribution of plausible transformations. In an RL context, we can also fine-tune the encoder parameters so that the latent space better aligns with states relevant to the environment.

**Latent Action Synthesis.** Here, the LPN encoder effectively synthesizes the *latent action* or *proto-action* by generating the embedding $z$, from which we derive $\hat{a}$. We bypass full DSL program generation. This adaptation yields a more powerful representation than a naive feed-forward network but can be computationally heavier due to the encoder's complexity (and potentially the sampling overhead if we retain the Bayesian sampling approach).

### 4.4.2 CNN + Multi-Head Self-Attention

To balance representational power with computational cost, we created a CNN-based pipeline inspired by ResNet-18, adding multi-head self-attention. Figure 4.4 illustrates the main components.

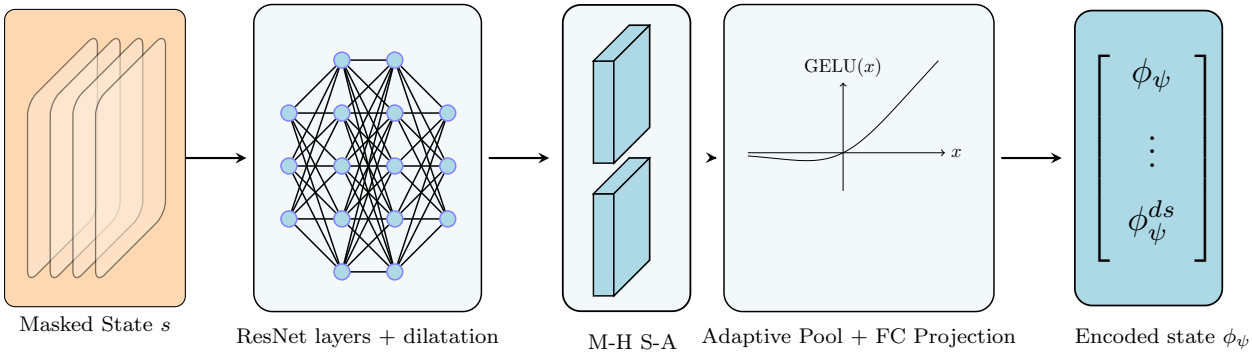



**Figure 4.4:** CNN with multi-head self-attention. After a modified ResNet stack processes the 4-channel input (2 original + 2 mask), we apply a multi-head attention block. We then adaptively pool and flatten, passing the result to a small FC layer. The final output is $\phi_s(s) \in \mathbb{R}^{d_s}$.

### 4.4.3 1. Channel Expansion with Masking

We begin with a grid $s \in \mathbb{R}^{H \times W \times 2}$ (2 channels for puzzle state). We create two additional mask channels (1.0 where a cell is valid, 0.0 otherwise), forming a $(4 \times H \times W)$ tensor. This is fed into a modified ResNet `conv1` (e.g., $4 \to 64$ kernels).

### 4.4.4 2. ResNet-like Convolutional Blocks

- **Layer 1, 2, 3:** Each consists of several residual blocks. We might remove `layer4` entirely to keep the feature map dimension moderate.
- **Dilation:** Convolutions in deeper blocks (layer2, layer3) can have dilation $(2, 2)$ to enlarge the receptive field without excessive downsampling.
- **Striding Adjustments:** If a block tries to downsample, we sometimes override its stride to maintain a consistent spatial resolution $(W' \times H')$.

### 4.4.5 3. Multi-Head Self-Attention

Whereas a single-head attention uses a single set of $(Q, K, V)$ transformations, **multi-head** attention divides channels (or embedding dimensions) into multiple heads that process queries, keys, and values in parallel. Suppose we have $N_{\text{heads}}$ heads; each head $h$ has:

$$Q_h = W_Q^h F, \quad K_h = W_K^h F, \quad V_h = W_V^h F,$$

where $W_Q^h, W_K^h, W_V^h$ map $C$ channels to $C_h = \frac{C}{N_{\text{heads}}}$ per head. For each head:

$$A_h = \text{softmax}\Big(\frac{Q_h^\top K_h}{\sqrt{C_h}}\Big), \quad O_h = V_h A_h^\top.$$

We then *concatenate* $O_1, \ldots, O_{N_{\text{heads}}}$ along the channel dimension, forming $\tilde{O} \in \mathbb{R}^{C \times (W'H')}$. Reshape back to $\mathbb{R}^{C \times W' \times H'}$. We optionally apply:

$$F_{\text{att}} = \gamma\, \text{Conv1x1}(\tilde{O}) + (1 - \gamma)\, F, \quad \text{LayerNorm}(F_{\text{att}}).$$

The $1 \times 1$ convolution merges heads if needed, and $\gamma$ is a learnable scalar for mixing the attended output and residual $F$.

### 4.4.6 4. Adaptive Pool + Fully Connected Projection

We apply AdaptiveAvgPool2d$(F_{\text{att}})$ to reduce $F_{\text{att}} \in \mathbb{R}^{C \times W' \times H'}$ to a fixed $(C \times 4 \times 4)$ shape. Flattening yields $16C$ features, which are passed through a linear layer:

$$\mathbf{z}_s = \text{GELU}\big(\text{Linear}(16C,\, d_s)\big),$$

possibly followed by dropout. This $\mathbf{z}_s \in \mathbb{R}^{d_s}$ is the final CNN embedding, i.e., $\phi_s(s)$. The actor's MLP uses $\phi_s(s)$ to produce $\hat{a} \in \mathbb{R}^d$, while the critic takes $\phi_s(s)$ as part of its state input.

# 5 Conclusions

Due to computational limitations, we were unable to train the model with the LPN encoder. Therefore, this section refers to the model utilizing a CNN and Multi-Head Attention as feature extractors. It is important to acknowledge that this architecture is suboptimal and lacks the capability to fully learn complex logical patterns. Moreover, the available computational resources were insufficient to train the model for enough episodes to ensure convergence.
Despite these constraints, our results provide evidence that the model exhibits learning capabilities in terms of both value loss and policy loss. The following figures illustrate the promising performance of an actor-critic network with approximately 25 million parameters. This represents the largest network we have trained, and our experiments indicate that increasing the model's scale significantly enhances its learning ability.

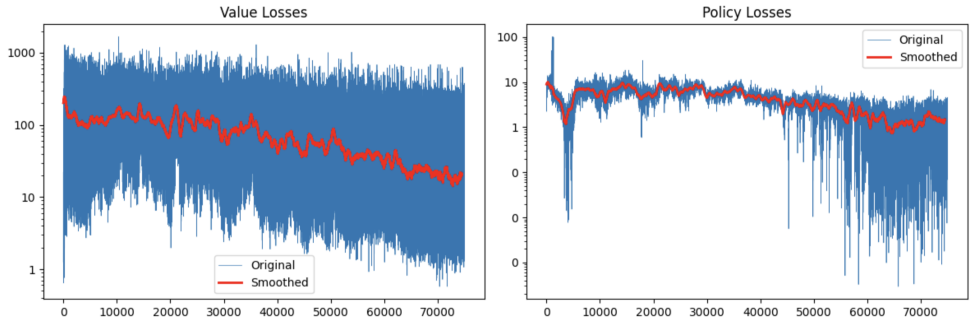



**Figure 5.1:** Training loss over time for value loss (left) and policy loss (right). The solid blue line represents the smoothed version of the loss, while the red line represents the original values. Both losses exhibit a decreasing trend, indicating learning progress.

The network was trained for approximately 80,000 steps, which we acknowledge is insufficient for full convergence to an optimal policy. Nevertheless, both value loss and policy loss show a steady decrease throughout training. Notably, the value loss decreases rapidly in logarithmic scale. Meanwhile, the decline in policy loss is slower but still significant, likely due to limitations in the action embeddings, as discussed in §5.1.3.

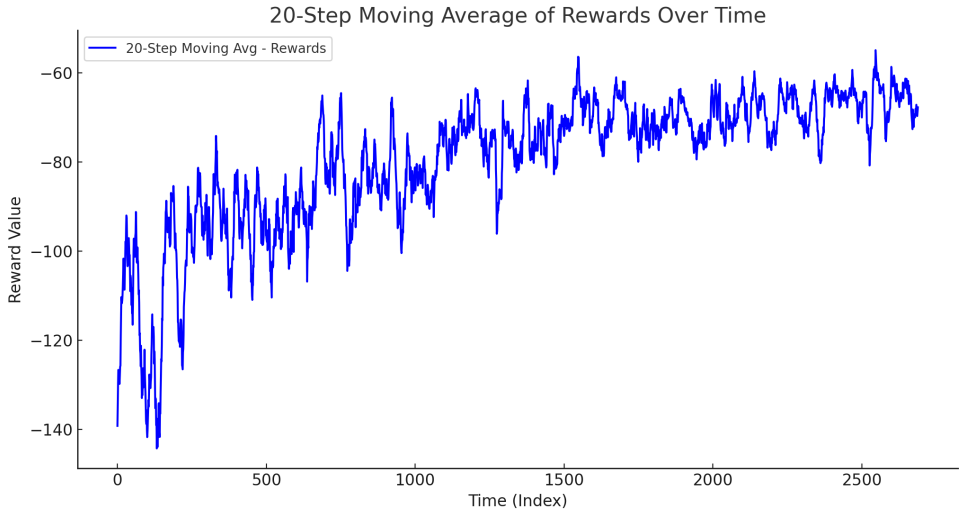



**Figure 5.2:** 20-step moving average of rewards over time. The increasing trend suggests that the model is progressively improving its ability to select better actions.

Encouragingly, the moving average of the episodic reward demonstrates a clear upward trend over time, indicating that the model is successfully improving its decision-making ability. This improvement is facilitated by the gradual reduction of the exploration factor (epsilon), which allows the model to prioritize learned policies over random exploration.

## 5.1 Limitations and Future Work

Although the proposed approach has not yet achieved the desired performance, there is considerable scope for refining and extending this initial model. In the following sections, we discuss the primary limitations encountered and propose directions for future research.

### 5.1.1 Computational Constraints

A major limitation of our study stems from the restricted computational resources available during experimentation. The limited hardware has constrained our ability to train larger networks and extend training durations. For instance, due to these constraints, we were unable to incorporate the LPN encoder (see Section 4.4.1) into our training regime—a modification that we believe could enhance the agent's environmental understanding. In the current setup, training was limited to approximately 10 hours on an NVIDIA P100 GPU. We expect that increased computational capacity will significantly improve the agent's performance.

### 5.1.2 Hierarchical Reinforcement Learning

The utilization of the Wolpertinger agent introduces inherent limitations, primarily because it represents actions as vectors in a $d$-dimensional space. This representation leads to two significant issues:

1. It fails to account for dependencies among the sub-actions within the $d$-dimensional action space.
2. The combinatorial nature of the domain-specific language (DSL) complicates scaling.

In our context, there exists a strong dependency among the three DSL function types (color selection, selection, and transformation). Constructing a meaningful action requires an understanding of the interdependencies among these functions. A promising direction is to develop a hierarchical model that first outputs a meta-action in a high-dimensional latent space of conceptual actions, and then sequentially constructs the final action (from selection to transformation). Such an approach could offer two main advantages:

1. It would more effectively capture the intricate dependencies between sub-actions.
2. It reduces the complexity of the decision process at each step, since the model would choose from a limited set of functions for each sub-action rather than generating the complete action in one step.

A critical challenge with this hierarchical method is the credit assignment problem. Since the environment provides a reward only for the overall action, it is unclear how to distribute this reward among the various sub-action components. An inappropriate allocation may inadvertently penalize a sub-action network for errors made by subsequent components.

### 5.1.3 Learnable Action Embeddings

Our current approach to action embeddings (see Section 3.1.2) is limited by its non-learnable nature, which restricts the agent's capacity to understand and navigate the action space effectively. At present, the similarity between actions is computed based on the similarity of their outputs. However, it would be more advantageous for the agent to autonomously learn the logical similarities between actions. For example, an action that turns the entire grid red and one that turns it yellow are considered distant in the current embedding space, despite their logical similarity. Future work should focus on developing a learnable embedding mechanism (either shared between or separate for the actor and critic) to improve action representation. One challenge with integrating a learnable embedding within the Wolpertinger framework is that the architecture relies on finding the $k$-nearest neighbors of a proto-action at each iteration. A dynamic embedding would necessitate reconstructing the nearest neighbor graph at every iteration, which could be computationally prohibitive. An alternative solution may involve combining a learnable embedding with a hierarchical reinforcement learning algorithm, as outlined above.

# 6 References

[1] François Chollet. On the measure of intelligence, 2019.

[2] Tursun Alkam and Ebrahim Tarshizi. Reinforcement learning in cognitive science: Foundations and applications. *Available at SSRN 5055984*.

[3] Ajay Subramanian, Sharad Chitlangia, and Veeky Baths. Psychological and neural evidence for reinforcement learning: A survey. *CoRR*, abs/2007.01099, 2020.

[4] Seungpil Lee, Woochang Sim, Donghyeon Shin, Wongyu Seo, Jiwon Park, Seokki Lee, Sanha Hwang, Sejin Kim, and Sundong Kim. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus, 2024.

[5] Michael Hodel. Domain-specific language for the abstraction and reasoning corpus.

[6] 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, 2019.

[7] 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, 2016.

[8] Clément Bonnet and Matthew V Macfarlane. Searching latent program spaces, 2024.

# A Appendix A: DSL

## A.1 Color Selection

The `ColorSelector` class provides a set of methods to select colors from a grid according to different strategies. In the context of the Abstraction Reasoning Corpus, these methods help determine colors based on their frequency or the size of their connected regions, facilitating grid-based reasoning.

### Most Common Color

```
def mostcolor(self, grid: np.ndarray) -> int:
    values = grid.flatten()
    counts = np.bincount(values, minlength=self.num_colors)
    return int(np.argmax(counts))
```

This method flattens the grid into a one-dimensional array, counts the occurrences of each color, and returns the color with the highest frequency.

### Least Common Color

```
def leastcolor(self, grid: np.ndarray) -> int:
    values = grid.flatten()
    counts = np.bincount(values, minlength=self.num_colors)
    counts[counts == 0] = self.big_number
    return int(np.argmin(counts))
```

Here the grid is flattened and color frequencies are computed. Zero counts are replaced with a large number to avoid selecting colors that do not appear, and the color with the lowest adjusted count is returned.

### Ranked Color Selection

```
def rankcolor(self, grid: np.ndarray, rank: int) -> int:
    unique_colors, counts = np.unique(grid, return_counts=True)
    sorted_indices = np.argsort(-counts)
    if rank < len(unique_colors):
        return int(unique_colors[sorted_indices[rank]])
    else:
        last_nonzero_index = sorted_indices[-1]
        return int(unique_colors[last_nonzero_index])
```

This function computes the frequency of each unique color, sorts them in descending order, and returns the color that ranks as the specified most common. If the provided rank exceeds the number of unique colors, the method defaults to returning the least common among those present.

### Rank of Largest Connected Shape

```
def rank_largest_shape_color_nodiag(self, grid: np.ndarray, rank: int) -> int:
    unique_colors = np.unique(grid)
    num_colors = len(unique_colors)
    dimension_of_biggest_shape = np.zeros(num_colors, dtype=int)
    for i, color in enumerate(unique_colors):
        color_mask = grid == color
        labeled_grid, _ = label(color_mask.astype(int))
        _, counts = np.unique(labeled_grid, return_counts=True)
        counts = counts[1:]
        dimension_of_biggest_shape[i] = np.max(counts) if counts.size > 0 else 0
    sorted_indices = np.argsort(-dimension_of_biggest_shape)
    if rank >= len(sorted_indices):
        smallest_nonzero_index = np.argmin(np.where(dimension_of_biggest_shape > 0,
    dimension_of_biggest_shape, np.inf))
        return int(unique_colors[smallest_nonzero_index]) if dimension_of_biggest_shape[
    smallest_nonzero_index] > 0 else int(unique_colors[0])
    return int(unique_colors[sorted_indices[rank]])
```

This method identifies connected components for each unique color (using non-diagonal connectivity) by labeling contiguous regions in a binary mask. It then determines the size of the largest connected region per color, sorts the colors by these sizes, and returns the color corresponding to the rank-th largest connected shape.
This function exists in two versions: with diagonal connections and without. The provided code does not consider diagonal connections.

## A.2 Selection

The `Selector` class implements a series of methods to extract and process regions in a 2D grid based on color and spatial connectivity within the Abstraction Reasoning Corpus. The class offers functionality to select cells matching a target color, extract rectangular regions composed solely of that color, identify connected regions using either 4- or 8-connectivity, and delineate both inner and outer borders of these regions. Each method returns a 3D boolean array where the first dimension indexes the different selections obtained.

### Single Color Mask

```python
def select_color(self, grid: np.ndarray, color: int) -> np.ndarray:
    mask = grid == color
    n_rows, n_cols = grid.shape
    mask = np.reshape(mask, (-1, n_rows, n_cols))
    return mask
```

This method creates a boolean mask by comparing each cell of the grid with the target color, reshapes the result into a 3D array with a single layer, and raises a warning if no cell in the grid matches the specified color.

### Rectangular Color Regions

```python
def select_rectangles(self, grid: np.ndarray, color: int, height: int, width: int) -> np.ndarray:
    rows, cols = grid.shape
    rectangles = []
    color_mask = self.select_color(grid, color)
    if np.sum(color_mask) == 0:
        return np.expand_dims(np.zeros_like(grid), axis=0)
    color_mask = color_mask[0, :, :]
    for i in range(rows - height + 1):
        for j in range(cols - width + 1):
            sub_rect = color_mask[i : i + height, j : j + width]
            if np.all(sub_rect):
                rect_mask = np.zeros_like(color_mask, dtype=bool)
                rect_mask[i : i + height, j : j + width] = True
                rectangles.append(rect_mask)
    if rectangles:
        result_3d = np.stack(rectangles, axis=0)
    else:
        result_3d = np.zeros((0, *color_mask.shape), dtype=bool)
    return result_3d
```

This function extracts rectangular regions of specified height and width that are completely filled with cells of the target color. It iterates over all valid starting positions in the grid, checks if the corresponding subgrid is entirely of the target color, and collects each valid rectangle as a separate layer in the resulting 3D array.

### Color Shapes with Edge Connections

```python
def select_connected_shapes(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_mask = self.select_color(grid, color)
    if np.sum(color_mask) == 0:
        return np.expand_dims(np.zeros_like(grid), axis=0)
    color_mask = color_mask[0, :, :]
    labeled_array, num_features = label(color_mask)
    shape = (num_features, *color_mask.shape)
    result_3d = np.zeros(shape, dtype=bool)
    for i in range(1, num_features + 1):
        result_3d[i - 1] = (labeled_array == i)
    return result_3d
```

This method isolates connected regions of the target color using 4-connectivity. It first obtains a boolean mask for the color, labels the connected components, and then extracts each component into its own layer in a 3D boolean array.

### Color Shapes with Edge and Corner Connections

```python
def select_connected_shapes_diag(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_mask = self.select_color(grid, color)
    if np.sum(color_mask) == 0:
        return np.expand_dims(np.zeros_like(grid), axis=0)
```

```
    color_mask = color_mask[0, :, :]
    structure = np.ones((3, 3), dtype=bool)
    labeled_array, num_features = label(color_mask, structure)
    shape = (num_features, *color_mask.shape)
    result_3d = np.zeros(shape, dtype=bool)
    for i in range(1, num_features + 1):
        result_3d[i - 1] = (labeled_array == i)
    return result_3d
```

This function operates like `select_connected_shapes` but uses an 8-connectivity structure (including diagonal neighbors) to label and extract connected regions of the specified color.

### Cells Adjacent to Color by Edges

```
def select_adjacent_to_color(self, grid: np.ndarray, color: int, points_of_contact: int)
    -> np.ndarray:
    nrows, ncols = grid.shape
    color_mask = self.select_color(grid, color)
    color_mask = color_mask[0, :, :]
    kernel = np.array([[0, 1, 0],
                       [1, 0, 1],
                       [0, 1, 0]])
    contact_count = convolve(color_mask.astype(int), kernel, mode="constant", cval=0)
    selection_mask = (contact_count == points_of_contact) & ~color_mask
    selection_mask = np.reshape(selection_mask, (-1, nrows, ncols))
    return selection_mask
```

This method identifies cells that are not of the target color but are adjacent (non-diagonally) to it with an exact number of contact points. It computes the number of neighboring cells of the target color using a cross-shaped convolution kernel and returns a mask where the contact count equals the specified value.

### Cells Adjacent to Color by Edges and Corners

```
def select_adjacent_to_color_diag(self, grid: np.ndarray, color: int, points_of_contact:
    int) -> np.ndarray:
    nrows, ncols = grid.shape
    color_mask = self.select_color(grid, color)
    color_mask = color_mask[0, :, :]
    kernel = np.ones((3, 3), dtype=bool)
    contact_count = convolve(color_mask.astype(int), kernel, mode="constant", cval=0)
    selection_mask = (contact_count == points_of_contact) & ~color_mask
    selection_mask = np.reshape(selection_mask, (-1, nrows, ncols))
    return selection_mask
```

This function extends the adjacent selection by considering all eight neighbors. It applies a full 3x3 convolution kernel to count adjacent cells of the target color and then returns a mask where cells (not of the target color) have exactly the given number of colored neighbors.

### Outer Borders of Edge-Connected Shapes

```
def select_outer_border(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_separated_shapes = self.select_connected_shapes(grid, color)
    for i in range(len(color_separated_shapes)):
        color_separated_shapes[i] = find_boundaries(color_separated_shapes[i], mode="
    outer")
    return color_separated_shapes
```

This method extracts the outer borders of connected shapes of the target color. It first identifies connected regions using 4-connectivity and then applies a boundary detection algorithm in ”outer” mode to each region.

### Inner Borders of Edge-Connected Shapes

```
def select_inner_border(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_separated_shapes = self.select_connected_shapes(grid, color)
    for i in range(len(color_separated_shapes)):
        color_separated_shapes[i] = find_boundaries(color_separated_shapes[i], mode="
    inner")
    return color_separated_shapes
```

This function selects the inner borders of connected shapes of the target color. It uses the 4-connectivity based segmentation and then extracts the inner boundaries by applying the boundary detection function

in ”inner” mode.

#### Outer Borders of Edge and Corner Connected Shapes

```
def select_outer_border_diag(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_separated_shapes = self.select_connected_shapes_diag(grid, color)
    for i in range(len(color_separated_shapes)):
        color_separated_shapes[i] = find_boundaries(color_separated_shapes[i], mode="
    outer")
    return color_separated_shapes
```

This method retrieves the outer borders of connected shapes identified with 8-connectivity. It first segments the grid into connected regions including diagonal neighbors and then extracts the outer boundaries for each region.

#### Inner Borders of Edge and Corner Connected Shapes

```
def select_inner_border_diag(self, grid: np.ndarray, color: int) -> np.ndarray:
    color_separated_shapes = self.select_connected_shapes_diag(grid, color)
    for i in range(len(color_separated_shapes)):
        color_separated_shapes[i] = find_boundaries(color_separated_shapes[i], mode="
    inner")
    return color_separated_shapes
```

This function mirrors `select_inner_border` but for shapes segmented using 8-connectivity. It extracts the inner borders of each connected region that includes diagonal connections.

#### Entire Grid

```
def select_all_grid(self, grid: np.ndarray, color: int = None) -> np.ndarray:
    nrows, ncols = grid.shape
    return np.ones((1, nrows, ncols), dtype=bool)
```

This method returns a mask covering the entire grid. It generates a 3D boolean array with a single layer where every cell is selected.

### A.3 Transformation

This class implements a variety of transformation methods on a grid structure for the Abstraction and Reasoning Corpus (ARC). It treats a 2D grid as a 3D array, where each “layer” corresponds to a binary mask of selected cells. Unless otherwise stated, each method returns a 3D version of the grid after applying the transformation.

#### Change Selected Cells to a New Color

```
def new_color(self, grid, selection, color):
    grid_3d = create_grid3d(grid, selection)
    grid_3d[selection == 1] = color
    return grid_3d
```

This function changes all selected cells to the specified color, ignoring color existence checks.

#### Apply a Specified Color to the Selection

```
def color(self, grid, selection, method, param):
    color_selected = select_color(grid, method, param)
    grid_3d = create_grid3d(grid, selection)
    grid_3d[selection == 1] = color_selected
    return grid_3d
```

This function determines a color via the given method (one among color selection methods) and parameter (for the color selection method), then assigns that color to every selected cell in the grid.

#### Fill Holes in a Single Connected Shape

```
def fill_with_color(self, grid, selection, method, param):
    grid_3d = create_grid3d(grid, selection)
    fill_color = select_color(grid, method, param)
    filled_masks = np.array([binary_fill_holes(i) for i in selection])
    new_masks = filled_masks & (~selection)
    grid_3d[new_masks] = fill_color
    return grid_3d
```

This function fills interior holes in the selected shape using binary hole-filling, then colors those newly filled cells.

### Fill the Bounding Rectangle with a Color

```
def fill_bounding_rectangle_with_color(self, grid, selection, method, param):
    color = select_color(grid, method, param)
    grid_3d = create_grid3d(grid, selection)
    bounding_rectangle = find_bounding_rectangle(selection)
    grid_3d = np.where(
        (bounding_rectangle & (bounding_rectangle & (1 - selection))) == 1,
        color,
        grid_3d
    )
    return grid_3d
```

This function identifies the smallest rectangle covering the selection and colors all of its empty cells.

### Fill the Bounding Square with a Color

```
def fill_bounding_square_with_color(self, grid, selection, method, param):
    color = select_color(grid, method, param)
    grid_3d = create_grid3d(grid, selection)
    bounding_square = find_bounding_square(selection)
    grid_3d = np.where(
        (bounding_square & (bounding_square & (1 - selection))) == 1,
        color,
        grid_3d
    )
    return grid_3d
```

This function finds the smallest square covering the selection and colors all of its empty cells.

### Flip Vertically Inside the Bounding Rectangle

```
def flipv(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    bounding_rectangle = find_bounding_rectangle(selection)
    flipped_bounding_rectangle = np.flip(bounding_rectangle, axis=1)
    grid_3d[bounding_rectangle] = np.flip(grid_3d, axis=1)[flipped_bounding_rectangle]
    return grid_3d
```

This function flips the selected area top-to-bottom within its bounding rectangle.

### Flip Horizontally Inside the Bounding Rectangle

```
def fliph(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    bounding_rectangle = find_bounding_rectangle(selection)
    flipped_bounding_rectangle = np.flip(bounding_rectangle, axis=2)
    grid_3d[bounding_rectangle] = np.flip(grid_3d, axis=2)[flipped_bounding_rectangle]
    return grid_3d
```

This function flips the selected area left-to-right within its bounding rectangle.

### Flip Along the Main Diagonal

```
def flip_main_diagonal(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    bounding_square = find_bounding_square(selection)
    for i in range(grid_3d.shape[0]):
        mask = bounding_square[i]
        rows, cols = np.where(mask)
        min_row, max_row = rows.min(), rows.max()
        min_col, max_col = cols.min(), cols.max()
        square = grid_3d[i, min_row:max_row + 1, min_col:max_col + 1]
        mirrored = square.T
        grid_3d[i, min_row:max_row + 1, min_col:max_col + 1] = mirrored
    return grid_3d
```

This function mirrors a square region along the main diagonal (top-left to bottom-right).

### Flip Along the Anti-Diagonal

```python
def flip_anti_diagonal(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    bounding_square = find_bounding_square(selection)
    for i in range(grid_3d.shape[0]):
        mask = bounding_square[i]
        rows, cols = np.where(mask)
        min_row, max_row = rows.min(), rows.max()
        min_col, max_col = cols.min(), cols.max()
        square = grid_3d[i, min_row:max_row + 1, min_col:max_col + 1].copy()
        mirrored = np.flip(np.rot90(square), 1)
        grid_3d[i, min_row:max_row + 1, min_col:max_col + 1] = mirrored
    return grid_3d
```

This function mirrors a square region along the anti-diagonal (top-right to bottom-left).

### Rotate a Square Region by Multiples of 90

```python
def rotate(self, grid, selection, num_rotations):
    grid_3d = create_grid3d(grid, selection)
    bounding_masks = find_bounding_square(selection)
    for i in range(bounding_masks.shape[0]):
        bounding_mask = bounding_masks[i]
        rows, cols = np.where(bounding_mask)
        row_start, row_end = rows.min(), rows.max() + 1
        col_start, col_end = cols.min(), cols.max() + 1
        sub_grid = grid_3d[i, row_start:row_end, col_start:col_end]
        rotated_sub_grid = np.rot90(sub_grid, num_rotations)
        grid_3d[i, row_start:row_end, col_start:col_end] = rotated_sub_grid
    return grid_3d

def rotate_90(self, grid, selection):
    return self.rotate(grid, selection, 1)

def rotate_180(self, grid, selection):
    return self.rotate(grid, selection, 2)

def rotate_270(self, grid, selection):
    return self.rotate(grid, selection, 3)
```

The first function rotates the selected square region 90 degrees counterclockwise `num_rotations` times. The following three functions are simple wrappers that call it with 1, 2, or 3 rotations respectively.

### Mirror the Selection Outside the Grid

```python
def mirror_down(self, grid, selection):
    d, rows, cols = selection.shape
    grid_3d = create_grid3d(grid, selection)
    new_grid_3d = np.zeros((d, rows * 2, cols))
    new_grid_3d[:, :rows, :] = grid_3d
    new_grid_3d[:, rows:, :] = np.flip(grid_3d, axis=1)
    flipped_selection = np.flip(selection, axis=1).astype(bool)
    new_grid_3d[:, rows:, :][~flipped_selection] = 0
    return new_grid_3d.astype(int)

def mirror_up(self, grid, selection):
    ...
    return new_grid_3d.astype(int)

def mirror_right(self, grid, selection):
    ...
    return new_grid_3d.astype(int)

def mirror_left(self, grid, selection):
    ...
    return new_grid_3d.astype(int)
```

These functions mirror the selection vertically or horizontally outside the original grid. Unmirrored cells in the extended region are set to zero.

### Duplicate the Selection Outside the Grid

```python
def duplicate_horizontally(self, grid, selection):
    d, rows, cols = selection.shape
```

```
    grid_3d = create_grid3d(grid, selection)
    new_grid_3d = np.zeros((d, rows, cols * 2))
    new_grid_3d[:, :, :cols] = grid_3d
    new_grid_3d[:, :, cols:][selection.astype(bool)] = grid_3d[selection.astype(bool)]
    return new_grid_3d

def duplicate_vertically(self, grid, selection):
    ...
    return new_grid_3d
```

These functions replicate the selected cells to a new region either on the right or below the current grid, ignoring any size checks.

### Copy-Paste Cells with Shifts

```
def copy_paste(self, grid, selection, shift_x, shift_y):
    grid_3d = create_grid3d(grid, selection)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    new_row_idxs = old_row_idxs + shift_y
    new_col_idxs = old_col_idxs + shift_x
    valid_mask = (
        (new_row_idxs >= 0) & (new_row_idxs < grid_3d.shape[1]) &
        (new_col_idxs >= 0) & (new_col_idxs < grid_3d.shape[2])
    )
    layer_idxs = layer_idxs[valid_mask]
    old_row_idxs = old_row_idxs[valid_mask]
    old_col_idxs = old_col_idxs[valid_mask]
    new_row_idxs = new_row_idxs[valid_mask]
    new_col_idxs = new_col_idxs[valid_mask]
    values = grid_3d[layer_idxs, old_row_idxs, old_col_idxs]
    grid_3d[layer_idxs, new_row_idxs, new_col_idxs] = values
    return grid_3d

def copy_sum(self, grid, selection, shift_x, shift_y):
    grid_3d = create_grid3d(grid, selection)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    new_row_idxs = old_row_idxs + shift_y
    new_col_idxs = old_col_idxs + shift_x
    valid_mask = (
        (new_row_idxs >= 0) & (new_row_idxs < grid_3d.shape[1]) &
        (new_col_idxs >= 0) & (new_col_idxs < grid_3d.shape[2])
    )
    layer_idxs = layer_idxs[valid_mask]
    old_row_idxs = old_row_idxs[valid_mask]
    old_col_idxs = old_col_idxs[valid_mask]
    new_row_idxs = new_row_idxs[valid_mask]
    new_col_idxs = new_col_idxs[valid_mask]
    values = grid_3d[layer_idxs, old_row_idxs, old_col_idxs]
    np.add.at(grid_3d, (layer_idxs, new_row_idxs, new_col_idxs), values)
    grid_3d = grid_3d % 10
    return grid_3d

def cut_paste(self, grid, selection, shift_x, shift_y):
    grid_3d = create_grid3d(grid, selection)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    new_row_idxs = old_row_idxs + shift_y
    new_col_idxs = old_col_idxs + shift_x
    valid_mask = (
        (new_row_idxs >= 0) & (new_row_idxs < grid_3d.shape[1]) &
        (new_col_idxs >= 0) & (new_col_idxs < grid_3d.shape[2])
    )
    values = grid_3d[layer_idxs[valid_mask], old_row_idxs[valid_mask], old_col_idxs[
    valid_mask]]
    grid_3d[layer_idxs, old_row_idxs, old_col_idxs] = 0
    grid_3d[layer_idxs[valid_mask], new_row_idxs[valid_mask], new_col_idxs[valid_mask]] =
     values
    return grid_3d

def cut_sum(self, grid, selection, shift_x, shift_y):
    grid_3d = create_grid3d(grid, selection)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    new_row_idxs = old_row_idxs + shift_y
    new_col_idxs = old_col_idxs + shift_x
    valid_mask = (
```

```
        (new_row_idxs >= 0) & (new_row_idxs < grid_3d.shape[1]) &
        (new_col_idxs >= 0) & (new_col_idxs < grid_3d.shape[2])
    )
    values = grid_3d[layer_idxs[valid_mask], old_row_idxs[valid_mask], old_col_idxs[
    valid_mask]]
    grid_3d[layer_idxs, old_row_idxs, old_col_idxs] = 0
    np.add.at(grid_3d, (layer_idxs[valid_mask], new_row_idxs[valid_mask], new_col_idxs[
    valid_mask]), values)
    grid_3d = grid_3d % 10
    return grid_3d
```

These functions shift the selected cells by (`shift_x`, `shift_y`). Copy-based methods keep the original intact; cut-based methods set the original location to 0. Summation variants combine overlapping values modulo 10.

### Copy-Paste Selection Vertically

```
def copy_paste_vertically(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    n_masks, height_of_grid, _ = grid_3d.shape
    rows_with_one = np.any(selection == 1, axis=2)
    first_rows = np.full(n_masks, -1)
    last_rows = np.full(n_masks, -1)
    for idx in range(n_masks):
        row_indices = np.where(rows_with_one[idx])[0]
        if row_indices.size > 0:
            first_rows[idx] = row_indices[0]
            last_rows[idx] = row_indices[-1]
    selection_height = last_rows - first_rows + 1
    factor_up = np.ceil(first_rows / selection_height).astype(int)
    factor_down = np.ceil((height_of_grid - last_rows - 1) / selection_height).astype(int
    )
    final_transformation = grid_3d.copy()
    for idx in range(n_masks):
        grid_layer = final_transformation[idx]
        selection_layer = selection[idx]
        grid_layer_3d = np.expand_dims(grid_layer, axis=0)
        selection_layer_3d = np.expand_dims(selection_layer, axis=0)
        for i in range(factor_up[idx]):
            shift = -(i + 1) * selection_height[idx]
            grid_layer_3d = self.copy_paste(grid_layer_3d, selection_layer_3d, 0, shift)
        for i in range(factor_down[idx]):
            shift = (i + 1) * selection_height[idx]
            grid_layer_3d = self.copy_paste(grid_layer_3d, selection_layer_3d, 0, shift)
        final_transformation[idx] = grid_layer_3d[0]
    return final_transformation

def copy_paste_horizontally(self, grid, selection):
    ...
    return final_transformation
```

These functions replicate the selected region upwards/downwards or leftwards/rightwards within the grid bounds as many times as possible, layering new copies without erasing the original.

### Gravitate the Whole Selection (Copy-Paste)

```
def gravitate_whole_downwards_paste(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    depth, rows, cols = selection.shape
    grid_without_selection = grid_3d.copy()
    indices = np.nonzero(selection)
    grid_without_selection[indices] = 0
    row_indices = np.arange(rows).reshape(1, rows, 1)
    sel_row_indices = np.where(selection, row_indices, -1)
    max_row_sel = sel_row_indices.max(axis=1)
    selection_exists_in_column = (max_row_sel != -1)
    max_row_sel_expanded = max_row_sel[:, None, :]
    mask_below_selection = row_indices > max_row_sel_expanded
    obstacles_below = (grid_without_selection != 0) & mask_below_selection
    obstacle_positions = np.where(obstacles_below, row_indices, rows)
    obstacle_positions = np.where(selection_exists_in_column[:, None, :],
    obstacle_positions, rows + 1)
    shift_per_column = obstacle_positions.min(axis=1) - max_row_sel - 1
```

```
    shift_per_depth = np.min(shift_per_column, axis=1)
    shift_per_depth = np.clip(shift_per_depth, 0, rows)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    shift_y = shift_per_depth[layer_idxs]
    grid_3d = self.copy_paste(grid_3d, selection, shift_x=0, shift_y=shift_y)
    return grid_3d

def gravitate_whole_upwards_paste(self, grid, selection):
    ...
    return grid_3d

def gravitate_whole_right_paste(self, grid, selection):
    ...
    return grid_3d

def gravitate_whole_left_paste(self, grid, selection):
    ...
    return grid_3d
```

These methods collectively move the entire selected region down, up, left, or right until it reaches a grid boundary or collides with non-zero cells. They replicate the selection rather than removing the original.

### Gravitate the Whole Selection (Cut-Paste)

```
def gravitate_whole_downwards_cut(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    depth, rows, cols = selection.shape
    grid_without_selection = grid_3d.copy()
    indices = np.nonzero(selection)
    grid_without_selection[indices] = 0
    row_indices = np.arange(rows).reshape(1, rows, 1)
    sel_row_indices = np.where(selection, row_indices, -1)
    max_row_sel = sel_row_indices.max(axis=1)
    selection_exists_in_column = (max_row_sel != -1)
    max_row_sel_expanded = max_row_sel[:, None, :]
    mask_below_selection = row_indices > max_row_sel_expanded
    obstacles_below = (grid_without_selection != 0) & mask_below_selection
    obstacle_positions = np.where(obstacles_below, row_indices, rows)
    obstacle_positions = np.where(selection_exists_in_column[:, None, :],
    obstacle_positions, rows + 1)
    shift_per_column = obstacle_positions.min(axis=1) - max_row_sel - 1
    shift_per_depth = np.min(shift_per_column, axis=1)
    shift_per_depth = np.clip(shift_per_depth, 0, rows)
    layer_idxs, old_row_idxs, old_col_idxs = np.where(selection)
    shift_y = shift_per_depth[layer_idxs]
    grid_3d = self.cut_paste(grid_3d, selection, shift_x=0, shift_y=shift_y)
    return grid_3d

def gravitate_whole_upwards_cut(self, grid, selection):
    ...
    return grid_3d

def gravitate_whole_right_cut(self, grid, selection):
    ...
    return grid_3d

def gravitate_whole_left_cut(self, grid, selection):
    ...
    return grid_3d
```

These methods are identical to the “paste” versions except they remove (cut) the original selected cells after shifting them, leaving the source region empty.

### Downward, Upward, Rightward, and Leftward Gravity (Per-Cell)

```
def down_gravity(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    num_layers, num_rows, num_cols = grid_3d.shape
    for layer_idx in range(num_layers):
        selection_layer = selection[layer_idx]
        selected_rows, selected_cols = np.where(selection_layer == 1)
        for i in range(len(selected_rows) - 1, -1, -1):
            row, col = selected_rows[i], selected_cols[i]
```

```
            value = grid_3d[layer_idx, row, col]
            grid_3d[layer_idx, row, col] = 0
            selection_layer[row, col] = 0
            for target_row in range(row + 1, num_rows):
                if grid_3d[layer_idx, target_row, col] != 0:
                    grid_3d[layer_idx, target_row - 1, col] = value
                    selection_layer[target_row - 1, col] = 1
                    break
            else:
                grid_3d[layer_idx, num_rows - 1, col] = value
                selection_layer[num_rows - 1, col] = 1
    return grid_3d

def up_gravity(self, grid, selection):
    ...
    return grid_3d

def right_gravity(self, grid, selection):
    ...
    return grid_3d

def left_gravity(self, grid, selection):
    ...
    return grid_3d
```

These transformations move each selected cell one at a time until it is blocked or reaches the boundary, thus "raining" them down, up, left, or right.

### Upscale the Selection Vertically & Horizontally

```
def vupscale(self, grid, selection, scale_factor):
    selection_3d_grid = create_grid3d(grid, selection)
    depth, original_rows, original_cols = selection.shape
    upscaled_selection = np.repeat(selection, scale_factor, axis=1)
    upscaled_selection_3d_grid = np.repeat(selection_3d_grid, scale_factor, axis=1)
    capped_selection = np.zeros((depth, original_rows, original_cols), dtype=bool)
    capped_upscaled_grid = np.zeros((depth, original_rows, original_cols))
    for layer_idx in range(depth):
        original_com = center_of_mass(selection[layer_idx])[0]
        upscaled_com = center_of_mass(upscaled_selection[layer_idx])[0]
        if original_rows % 2 == 0:
            half_rows_top, half_rows_bottom = original_rows // 2, original_rows // 2
        else:
            half_rows_top, half_rows_bottom = original_rows // 2 + 1, original_rows // 2
        lower_bound = min(int(upscaled_com + half_rows_bottom), original_rows *
    scale_factor)
        upper_bound = max(int(upscaled_com - half_rows_top), 0)
        capped_selection[layer_idx] = upscaled_selection[layer_idx, upper_bound:
    lower_bound, :]
        capped_upscaled_grid[layer_idx] = upscaled_selection_3d_grid[layer_idx,
    upper_bound:lower_bound, :]
        capped_com = center_of_mass(capped_selection[layer_idx])[0]
        offset = capped_com - original_com
        lower_bound += offset
        upper_bound += offset
        if lower_bound >= original_rows * scale_factor:
            lower_bound = original_rows * scale_factor
            upper_bound = lower_bound - original_rows
        elif upper_bound <= 0:
            upper_bound = 0
            lower_bound = upper_bound + original_rows
        capped_selection[layer_idx] = upscaled_selection[layer_idx, upper_bound:
    lower_bound, :]
        capped_upscaled_grid[layer_idx] = upscaled_selection_3d_grid[layer_idx,
    upper_bound:lower_bound, :]
    selection_3d_grid[selection == 1] = 0
    selection_3d_grid[capped_selection] = capped_upscaled_grid[capped_selection].ravel()
    return selection_3d_grid

def hupscale(self, grid, selection, scale_factor):
    ...
    return selection_3d_grid
```

This function vertically repeats each selected row according to `scale_factor`, then centers and caps the result to the original bounding rows.

### Crop to the Selection's Bounding Rectangle

```
def crop(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    bounding_rectangle = find_bounding_rectangle(selection)
    for i in range(selection.shape[0]):
        bounding_rectangle[i] = np.ones_like(grid_3d[i], dtype=bool) if not
    bounding_rectangle[i].any() else bounding_rectangle[i]
    grid_3d[~bounding_rectangle] = -1
    return grid_3d
```

This function zeros out every cell outside the bounding rectangle around the selection by setting it to $-1$.

### Delete Selected Cells

```
def delete(self, grid, selection):
    grid_3d = create_grid3d(grid, selection)
    grid_3d[selection] = 0
    return grid_3d
```

This function simply sets any selected cell to 0.