Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Program synthesis on the Abstraction Reasoning Corpus with Large Language Models

Semester Thesis

Julian Kleutgens
jkleutgens@ethz.ch

Institute of Neuroinformatics
ETH Zürich

Supervisors:
M. Sc. Yassine Taoudi-Benchekroun
Prof. Dr. Benjamin Grewe

June 21, 2024

Abstract

Humans can generalize sparse experiences into principles and apply them in unfamiliar contexts. In contrast, artificial agents often struggle in novel situations, even with large amounts of data. François Chollet argues this happens because artificial agents are trained to be skillful rather than genuinely intelligent. To address this and steer research toward true AI, he introduced the Abstraction and Reasoning Corpus (ARC), a benchmark designed to measure general intelligence. The ARC dataset evaluates a system's ability to deduce rules from limited input-output pairs and apply them to new, unseen data.

A key contribution of this thesis is developing a methodology using a Domain-Specific Language (DSL) to represent solutions for ARC tasks. This involves training Large Language Models (LLMs) to generate DSL programs based on ARC dataset input-output pairs. Initially, the thesis discusses the motivation and background for measuring intelligence in AI and explains Michael Hodel's DSL, showcasing its utility in solving ARC tasks.

Next, the thesis explores the implementation and performance of a transformer-based model named Stormer for multi-label classification of DSL functions. Recognizing this approach's limitations, the research shifts to using the T5 model, a pre-trained LLM, to generate DSL solvers.

Results show the T5 model can learn to generate correct and generalizable DSL solvers, though challenges remain in achieving high accuracy and consistent output generation.

Contents

Chapter 1

Introduction

1.1 Motivation

In this section, I will discuss the motivation and intentions behind this thesis. As the term "Artificial Intelligence" suggests, it involves building an intelligent machine, but defining and measuring this attribute is challenging. In the paper "On the Measure of Intelligence," the author François Chollet discusses current research in machine learning and addresses the problem of measuring intelligence in both human psychology and machine learning [1]. Current datasets or benchmarks, such as ImageNet, primarily focus on measuring "skill" or "performance" on specific tasks. The paper argues that evaluating skill alone is insufficient for measuring intelligence, as skill is heavily influenced by prior knowledge and experience. With unlimited prior information or training data, experimenters can artificially boost a system's skill level, obscuring its true ability to generalize. This issue is particularly pronounced with humans. The general intelligence measures like IQ are still debated among experts, but it is widespread agreement on how to measure performance in specific tasks/skills like chess, running, or speed-reading.

Therefore, intelligence should first be seen in broad or general-purpose abilities (skill acquisition and generalization) rather than in skill alone. It should also consider experience and priors and attempt to quantify the strength of generalization. This leads to the following definition:

"The intelligence of a system is a measure of its skill-acquisition efficiency over a scope of tasks, with respect to **priors**, experience, and **generalization** difficulty."

1.1.1 Priors and Generalization

The definition of **priors** in a human sense, as discussed in the paper, comes in multiple forms. There are low-level priors about the structure of our senses and our built-in motor responses. Meta-learning priors shape how we learn itself. For

instance, the concept of causality, where one event leads to another, is something we inherently understand, helping us figure out how the world works. There are also high-level knowledge priors related to objects and phenomena in our external environment, which help us quickly understand objects, navigate space, and interpret the behavior of others.

For the goal of building human-like artificial intelligence, focusing on low-level sensorimotor instincts isn't necessary unless we're also trying to replicate a human body. However, human meta-learning priors (how the brain learns and adapts) are of the highest interest.
Generalization is the ability of a system to apply what it has learned from a set of prior experiences to new and different situations. Artificial generalization can be put in different perspectives:

1. **System-centric generalization**: This refers to a system's capacity to tackle new problems or scenarios, even if it hasn't specifically been trained or programmed for them.

2. **Developer-aware generalization**: This takes things further. This is the ability of a system, either learning or static, to handle situations that neither the system nor the developer of the system has encountered before. While including the developer in the system would be equivalent to "system-centric generalization"

The goal of machine learning is to achieve Developer-aware generalization for it to handle correctly in situations that even the developer has not thought of. Additionally, Chollet defines different degrees of generalization for information-processing systems:

1. **Absence of Generalization:** Generalization relies on novelty and uncertainty. A system can't generalize if it lacks uncertainty. For example, a tic-tac-toe program using exhaustive iteration or a proven sorting algorithm doesn't generalize, as they cover all possible scenarios by definition. (Hardcoded code vs relying on probabilities for next best move)

2. **Local Generalization (Robustness):** Systems that can handle new, but similar, data points within a known task, like an image classifier trained on labeled images. This is the main focus of machine learning historically, involving adaptation to known unknowns within a single task.

3. **Broad Generalization (Flexibility):** Systems capable of handling a wide range of tasks and unexpected situations without human intervention, like a self-driving car driving in a complete new environment.

4. **Extreme Generalization:** This is the capacity to handle entirely new tasks across a wide scope, sharing only abstract similarities with previous

experiences. Only biological intelligence, like humans, exhibits this. A subset of this, "human-centric extreme generalization" or "generality," refers to tasks within human experience, showing adaptability to highly novel situations and those never encountered before in evolution.

Many datasets and tasks for e.g. in Computer Vision are merely focusing on Local Generalization. For going a step further into Broad Generalization, they purpose, what to expect of an ideal intelligence benchmark properties that a candidate benchmark of human-like general intelligence should possess.

1.1.2 Benchmark Evaluation Criteria

To effectively evaluate AI systems, a benchmark must meet several key criteria:

- **Scope and Predictiveness**: The benchmark should clearly define its application scope and demonstrate its predictive validity. This involves empirically showing a strong statistical correlation between success on the benchmark and success on various real-world tasks.
- **Reliability**: The benchmark must be reproducible. Any stochastic elements in the evaluation should not significantly affect the outcomes.
- **Broad Abilities and Generalization**: It should assess a wide range of abilities, not just specific skills or the highest potential performance. The tasks should be unknown in advance to both the system and its developers, ensuring fair evaluation (**developer-aware generalization**). It should clearly specify the type of generalization it measures, such as local (robustness), broad (flexibility), or extreme (general intelligence), to prevent exploiting shortcuts.
- **Control of Experience**: The benchmark should limit the amount of experience that systems can use during training. This prevents systems from gaining an advantage by accessing unlimited training data. Ideally, the tasks should be such that new data cannot be easily generated for practice.
- **Explicit Priors:** The priors should be explicitly describe, which are assumable involved. This transparency prevents hidden biases that could unfairly benefit either humans or machines.
- **Fairness for Humans and Machines:** The benchmark should be designed to be fair for both humans and machines. It should rely only on basic human knowledge (Core Knowledge) and require a reasonable amount of practice or training data, comparable to what a human would need.

Within this spirit to go a step further, Chollet proposed a benchmark data set: the **Abstraction and Reasoning Corpus** (ARC). With that dataset, they want to challenge the AI into **Developer-aware generalization** and **Broad Generalization**. This brings me to the introduction of the ARC dataset.

1.2 Abstraction and Reasoning Corpus

In this section, I will briefly explain the dataset and its design. Additionally, I will detail the dataset's properties, particularly in relation to subsection 1.1.2. The ARC dataset comprises a training set of 400 tasks and an evaluation set of 600 tasks. The evaluation set is further split into a 400-task public evaluation set and a 200-task private evaluation set. This differs from common datasets, which typically have a 90%/10% training/testing data split. Each task (or sample) in the dataset consists of a few input/output grid pairs (usually 3 to 5). The goal is to deduce the "rule" that systematically transforms the input grid into the output grid. This discovered rule is then applied to the test input grid within that task to generate the corresponding output grid. With only one input/output pair, it's nearly impossible to discern the rule. This is either due to a lack of confirmation in other pairs or an overwhelming number of possible rule applications. All tasks are unique and follow their own rules.

A grid can be any height or width between 1x1 and 30x30 with 10 possible colors. They are not necessarily square (the median height is 9 and the median width is 10). The only feedback provided by the test (whether taken by a human or a machine) is binary (correct or incorrect).

The existence of a private evaluation set with its own unique rules aims to measure "developer aware generalization," as previously described. A test taker is assumed to have full access to the training set. Even without prior knowledge, a human test taker can solve most of the ARC evaluation set without any previous training. However, machines can seamlessly perceive tasks as structured matrices, enabling them to perform mathematical operations. Thus, the training set can be used to familiarize an algorithm with the content of Core Knowledge priors. The following Core Knowledge priors must be learned to create a fair comparison between human intelligence and artificial intelligence:

1. **Objectness:**
    - Objects are distinct entities within the grid. Separate objects from the background color.
    - Objects persist despite minor variations or the presence of noise.
    - Objects can interact upon contact.
2. **Goal-Directedness:**

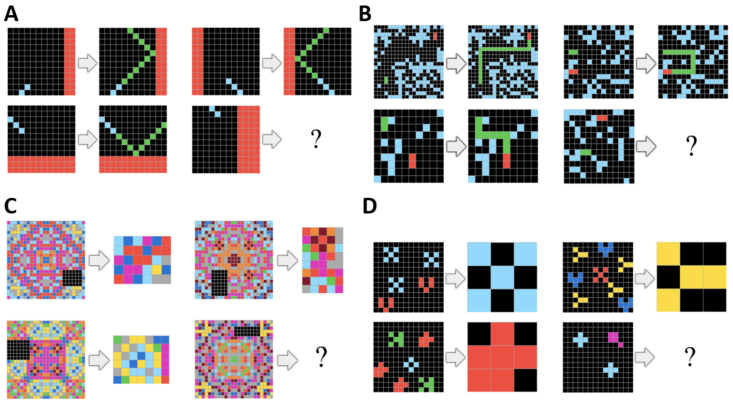


Figure 1.1: Here are four example tasks that focus on different core knowledge principles: A) Extrapolating a Diagonal Line with Obstacle Interaction: A task where the implicit goal is to extrapolate a diagonal line that "rebounds" upon contact with a red obstacle. B) Combining Multiple Concepts: A task that combines the concepts of "line extrapolation," "avoiding an obstacle," and "efficiently reaching a goal." C) Completing a Symmetrical Pattern: A task where the implicit goal is to complete a symmetrical pattern. The nature of the task is specified by three input/output examples. D) Counting and Selecting Unique Objects: A task where the implicit goal is to count unique objects and select the object that appears most frequently. This was obtained from the Master Thesis of Yassine Taoudi Benchekroun.

- Input and output grids often represent a purposeful transformation.
- The tasks don't have the concept of time, but can be seen as initial state and output state.

3. **Numbers & Counting:**
   - The ARC tasks often involve counting, sorting, and comparing objects based on size or quantity.
   - Tasks may also involve repeating patterns or basic addition and subtraction with numbers less than 10. An example task would be Figure 1.1 D).
4. **Basic Geometry & Topology:**
   - Shapes, lines, and patterns are significant. An example task would be Figure 1.1 A).
   - Tasks may involve transformations (e.g., rotation, scaling) or spatial relationships (e.g., inside/outside).

In Figure 1.1 are examples of learning the Core Knowledge priors. In theory, if the machine learns these priors, a fair comparison between human and artificial

intelligence can be drawn. Additionally, it would address the question of whether a machine is capable of meta-learning and generalization. The author suggests that current machine learning methods are not well-suited for this task. He believes that ARC should be approached as a program synthesis challenge, likely involving a Domain-Specific Language (DSL), rather than using deep learning, which I will introduce in Chapter 3.

1.3 Agenda

The focus of this thesis is to solve the ARC dataset using Large Language Models (LLM). The LLM is employed to generate the Domain-Specific Language (DSL) solver that then addresses the input-output pairs. To achieve this, the following topics will be explored:

1. **Related Work**: A review of two papers that tackle issues of generalized learning and meta-learning in LLMs, although they do not use the ARC dataset. Additionally, the paper "CodeIT," which closely relates to this thesis and also uses the ARC dataset will be covered.

2. **Domain-Specific Language**: Chapter 3 introduces the DSL by Michael Hodel, used as a solver for the tasks. This section also covers the data generation of prior knowledge for the training data. Furthermore, the distribution of generated solvers and the variance in the generated data will be examined.

3. **Classification task on DSL**: The background and theory of LLMs and the Transformer architecture will be explained. This section will also discuss experiments conducted on a self-built encoder architecture, "Stormer," and the results obtained.

4. **Pre-trained Model T5**: The core of this thesis involves explaining the pre-trained model T5. Attempts to solve the ARC with T5, including various modifications to the representation of the solver and tasks, and the tokenization process, will be described. Additionally, the metrics used to measure the results will be explored. In the end, the results and potential future work to improve or further test the model will be discussed.

Chapter 2

Related Work

In the introduction, it is established that intelligence should be measured by the efficiency of skill acquisition and generalization, rather than just task-specific performance. Current AI models have significant limitations in systematic reasoning and compositional tasks. Other research papers also aim to advance our understanding of intelligence, both human and artificial. These papers focus on defining, measuring, and comparing intelligence to enable the development of more human-like artificial systems.

The first two papers share the same goal as the ARC challenge but use simpler datasets without prior knowledge training. They apply transformer architecture-based models directly to their defined tasks and conclude that the capability of machine learning models to generalize.

The third paper closely aligns with the objectives of this thesis, they also tackle the ARC Challenge with the DSL.

2.1 Faith and Fate: Limits of Transformers on Compositionality

The paper "Faith and Fate: Limits of Transformers on Compositionality" [2] investigates the limitations of transformer LLMs in performing complex multi-step reasoning tasks. The authors focus on three representative compositional tasks: multi-digit multiplication, logic grid puzzles, and a dynamic programming problem. For evaluation, GPT-3, ChatGPT (GPT-3.5-turbo), and GPT-4 were tested using zero-shot, few-shot, and fine-tuning techniques. Scratchpads, which are verbalizations of computation graphs, were used to get a reasoned enumerated answer. Both question-answer and question-scratchpad formats were considered for few-shot and fine-tuning settings.

The visualization of multi-digit multiplication into a graph can be seen on Figure 2.1. Note that they did the experiments by prompting, so they described a multiplication task in words for the LLM to correctly follow and enumerate

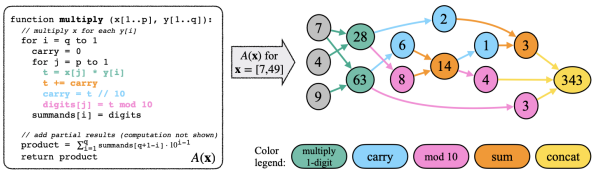


Figure 2.1: Transformation of an algorithm A to its computational graph. The depicted example is of long-form multiplication algorithm A, for inputs x = [7, 49].

the steps. The study classified nodes into four categories: fully correct (ancestors have correct values), local error (parent nodes have correct values, but the current node is wrong), propagation error (derived from a correct computation but some of its parent nodes are incorrect), and restoration error (has a correct value but is derived from an incorrect computation). Results showed that the ratio of fully correct nodes decreased with increasing graph layers, while propagation errors were higher than local errors. This indicates that models perform single-step reasoning well but struggle with multi-step compositional reasoning. High restoration errors in certain tasks suggest memorization of training data patterns rather than true reasoning.

The results of this paper show that LLM excels in single-step operations, but their performance degrades with increased reasoning depth. The prevalence of restoration errors indicates that these models often rely on memorized patterns from training data, rather than genuine multi-step reasoning capabilities. This dependence on **memorization sabotages their ability to generalize to new**, unseen problems, pointing to the need for more advanced techniques to enhance their reasoning abilities.

2.2 Human-like Systematic Generalization through a Meta-Learning Neural Network

This article addresses Fodor and Pylyshyn's challenge by providing evidence that neural networks can achieve human-like systematicity when optimized for their compositional skills. The authors introduce the Meta-Learning for Compositionality (MLC) approach, which guides training through a dynamic stream of compositional tasks. MLC is an optimization procedure that enhances a neural network's ability to generalize systematically.

The study uses a standard seq2seq transformer architecture with an encoder and decoder, each consisting of 3 layers, 8 attention heads per layer, 128-sized embeddings, and a feedforward hidden size of 512. The network is trained to process and generate sequences from combined input and study examples.

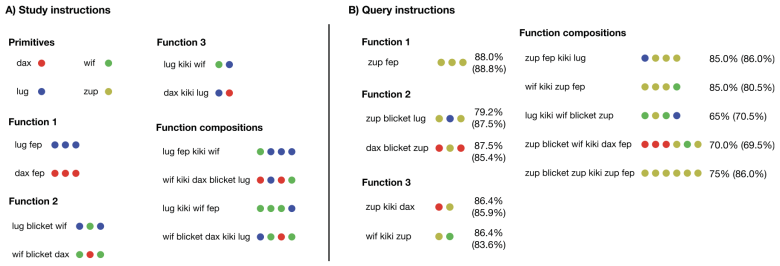


Figure 2.2: Few-shot instruction learning task with a full set of queries. Few-shot instruction learning task with a full set of queries.

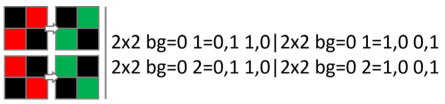
The MLC approach involves an optimization procedure that encourages systematicity through a series of few-shot compositional tasks. For example, in Figure 2.2, the goal is to predict a query sequence (B) based on given instructions (A). The colors and names switch with each task. Both humans and the machine were tested on these tasks. MLC is designed to achieve high accuracy while also mimicking human-like error patterns, reflecting inductive biases observed in human learners. Results show that MLC can mimic human performance, demonstrating lower error rates and producing human-like error patterns. MLC balances systematicity and inductive biases effectively, defining inductive bias as mapping each input word to exactly one output symbol. It exhibits stronger systematicity than standard neural networks and better predicts human behavior. However, MLC **struggles with unpracticed forms of generalization**, handling entirely novel structures, and generating new symbols.

2.3 CodeIt: Self-Improving Language Models with Prioritized Hindsight Replay

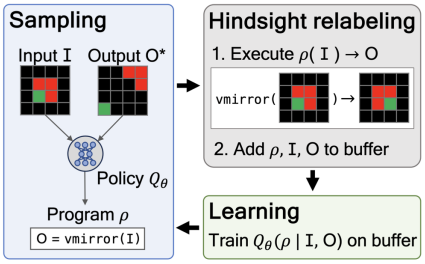
The authors propose a method for generating new tasks by predicting the Solver DSL (Domain-Specific Language) function and solving the task itself [3]. The setup, as shown in Figure 2.3b, involves using 400 DSL solvers from the original ARC training set as ρ. For predicting the DSL solver function ρ given the input-output grids, they use a policy $Q_e(\rho|I, O)$. This policy is implemented using a pretrained encoder-decoder Large Language Model (LLM), specifically the 220 million parameter CodeT5+. They train the T5 model with a specified loss function.

$$\mathcal{L}(\rho, I, O) = -\log Q_\theta(\rho \mid I, O). \tag{2.1}$$

The Code Iteration Algorithm begins with the initialization phase, where a dataset of ARC training tasks and solution programs written in a domain-specific language (DSL) is established. This dataset undergoes expansion through the



(a) Sparse grid representation of a simplified ARC task.



(b) An overview of Code Iteration.

Figure 2.3

random mutation of programs, effectively augmenting the training set. This augmentation serves as a form of data enrichment, enabling the model to learn the DSL syntax even from a limited training set.

In the sampling and hindsight relabeling stage, new programs are generated using the policy Q_θ. For each task, the input and target outputs are transformed from grids into textual format, encoded, and subsequently decoded into a program via the policy. The generated program is executed on the input grids. Programs that are syntactically incorrect or excessively slow are discarded. Conversely, if a program executes correctly, it is added to a replay buffer along with the input and output. This cycle is repeated multiple times for each task.

During the learning phase, the policy Q_θ undergoes training on experiences pulled from the replay buffer, the original training set, and the augmented training set. This iterative process enhances the model's ability to generalize and effectively learn the DSL syntax. For the Grid representation shown in Figure 2.3a. Each color is encoded as an integer, and for each color in the grid we list all the grid cells with that color as $[x, y]$ coordinates, so the background colors positions are omitted in the representation. It works works well for sparse grids and is human-interpretable.

With that code they solve 59 of 400 evaluation tasks.

Chapter 3

Domain-Specific Language: Framework and Task Generation for ARC

3.1 Domain Specific Language (DSL)

The goal is to develop a Domain-Specific Language (DSL) capable of representing solution programs for any ARC task **??**. By combining various components of the DSL, we can generate potential solution programs. The primary aim is to establish a solid foundation for future ARC research by creating a concise DSL that enables the creation of short, efficient programs to solve ARC tasks. The motivation behind this effort is the belief that such a DSL will provide essential groundwork for future advancements. This DSL was designed and developed by Michael Holder. The DSL should meet two main objectives:

1. It must be expressive enough to describe programs for most ARC tasks uniquely within the DSL. This is crucial because a comprehensive ARC language needs this level of expressiveness.

2. It should be abstract and generic, using a small set of primitives that are widely applicable across various ARC tasks. This approach minimizes overfitting to specific training tasks and enables the creation of concise solution programs, which is essential for any program synthesis method.

The DSL consists of 160 functions and 27 constants. The number of arguments for a DSL function varies and can include constants as inputs. These constants are recognizable by being written in capital letters. What makes the DSL special is that it does not require any if-else statements or for loops; instead, each function takes a number of arguments, and the result is assigned to a variable. Variables are labeled with `x` followed by a number until the result is achieved. The DSL is an effective way of solving ARC tasks because each task has its own unique "rule," which can be perfectly described by the DSL.

Michael wrote the solvers by hand for the first 400 training data tasks. While one could write the solvers by hand for the evaluation set as well, this would lead to **Absence of Generalization**, which is not the goal of the ARC challenge, as explained in subsection 1.1.1.

The 160 DSL functions can be categorized into three roughly equally sized categories of primitives:

1. **Transformations**: Functions that transform a grid or an object, e.g., resize, mirror, rotate, recolor, move, delete, etc.
2. **Properties**: Functions that extract features of an entity, e.g., leftmost occupied cell, center of mass, shape, size, whether an object is a line, a square, etc.
3. **Utils**: Functions that implement set operations (e.g., difference, union, intersection, insertion, removal), arithmetic operations (e.g., addition, subtraction, multiplication, floor division), or provide various helper functionalities (e.g., filtering, function composition, parameter binding, branching, merging of containers).

Let's go through an example solver to make sense of and get an idea of the DSL. The example would be for task `00d62c1b`, which can be found in the appendix section A.1. The rule of the task is to fill in all the holes of an object with yellow pixels. The solver for this task is:

```
def solve_00d62c1b(I):
    x1 = objects(I, T, F, F)
    x2 = colorfilter(x1, ZERO)
    x3 = rbind(bordering, I)
    x4 = compose(flip, x3)
    x5 = mfilter(x2, x4)
    O = fill(I, FOUR, x5)
    return O
```

1. Let x_1 be the set of objects extracted from the input grid I that are single-colored only. These objects must have cells that are directly connected and can include cells of the background color (black). This set is the result of calling the primitive objects function on I with the parameters `univalued=True` (Specifies if cells in an object must be of a single color), `diagonal=False` (Indicates if a cell must be directly adjacent to belong to an object, or if diagonal adjacency is allowed), and `without_background=True` (Determines if background-colored cells, the most common color, are included in objects or ignored.).

2. Let x_2 be the subset of x_1 consisting of objects that are black. Takes the constant `color=ZERO` (black) and x1 as an argument.

3. Define x_3 as a function with the signature f : object $\rightarrow$ bool that returns `True` if and only if an object is at the border of the grid.

4. Let x_4 be a function that returns the inverse of x_3. Specifically, x_4 returns `True` if and only if an object does not border the grid. This function is the result of composing the `flip` primitive, which negates a boolean value, with x_3. Compose can be viewed like $compose(f, g) \implies f(g(x))$.

5. Let x_5 be a single object defined as the union of objects in x_2 for which the function x_4 returns `True`. These are the black objects that do not border the grid (corresponding to the "holes" in the green objects). This object is created by calling the `mfilter` function, which combines merging and filtering, with the parameters `container=x2` and `condition=x4`.

6. Finally, let the output grid x_6 be created by coloring all pixels of the object x_5 yellow. This is achieved by calling the `fill` primitive on I with the parameters `color=FOUR` (yellow) and `patch=x5`.

This information can be found in the write-up from Michael's repository: [Michael's ARC-DSL Repository] (https://github.com/michaelhodel/arc-dsl/tree/main). With the DSL explained, this brings me to how Micheal can generate artificial ARC tasks.

3.2 Data Generation

As explained in section 1.2, we must train the machine on prior knowledge, such as Goal-Directedness and Objectness, and then apply the trained skills to the ARC challenge itself. For this initial step, Michael also worked on data generation using his defined DSL. Specifically, he focused on generating tasks that can vary in the number of input-output pairs. Each task applies one rule to the input grid, resulting in the corresponding output grid, just like the real ARC dataset.

Typically, modern data generation relies on deep learning. For instance, text generation utilizes transformer architecture-based LLMs, and Diffusion models are state-of-the-art for images. Task generation with deep learning would involve a two-player game, where a generator produces the task, and a discriminator determines if it is a valid, solvable task with a common rule applied. In the original paper "On the Measure of Intelligence" the author suggests generating tasks through a continuous learning program referred to as a "teacher" program, which interacts in a loop with test-taking systems called "student" programs. The teacher program optimizes task generation to ensure novelty and interest for a given student, making tasks new and challenging yet still solvable by the

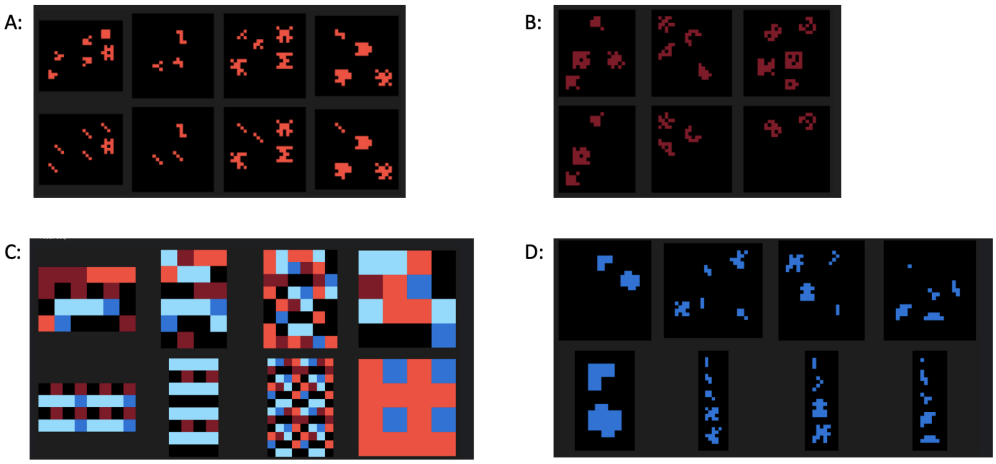


Figure 3.1: This figure provides examples of the generated data. **A:** This example is from the schema "Conditional Transform," where the rule is to convert the object to a diagonal line if it satisfies the condition of having a height of three. **B:** Also from "Conditional Transform," this rule requires rotating the object three times if it satisfies the condition of being diagonally symmetric and removing all objects that do not meet this condition. **C:** From the schema "Grid Level," this rule involves cutting out the inner center—removing all pixels not at the grid's border—and repeating it four times. **D:** This example is from the schema "Order Relation." The rule here is to arrange all objects in ascending order based on their width and place them vertically on a new grid with a padding of 2.

student. Meanwhile, students evolve to learn how to solve increasingly difficult tasks. This arrangement is beneficial for curriculum optimization, as the teacher program can be configured to enhance the learning efficiency of its students. This concept is similar to the POET system introduced in [4], which would include deep reinforcement learning.

However, for the first approach, Michael employs a more direct method by defining a fixed number of rules or **Schematas** described as a set of DSL Solvers. These solvers are then applied to randomly initialized input grids to receive an output grid. In the second approach, he uses the technique of **Reverse Engineering** [5].

Let's delve deeper into the first method. Michael defines three different **Schematas** for the applied rules: Conditional Transform, Grid Level, and Order Relations. Unfortunately, this is not yet documented, so the only reference is the Python code itself.

1. **Conditional Transform**: There are 51 different **conditions** and 50 different **transformation** types defined, with their combinations forming a

rule. This rule is applied to objects that have a different color as the background. In Figure 3.1, A and B demonstrate this. A dictionary is used to map both conditions and transformations with the DSL solvers. For example, the condition *Rectangle* corresponds to the string *"fork(equality, size, fork(multiply, height, width))"*. This expands in the solver to: *x1 = fork(multiply, height, width)* and *x2 = fork(equality, size, x1)*. To introduce more variance, tasks differ by either **removing** all objects that do not satisfy the condition (T) or keeping them (F) in the output grid. Additionally, three different **object augmentations** are generated: changing the color (SR), maintaining the color (ST), or painting the transformed objects on top of the unchanged original object (OP). The total number of different rules can be calculated as:Conditions $\times$ Transforms $\times$ Removing $\times$ Object Augmentation $\times$ Colors $= 51 \times 50 \times 2 \times 3 \times 10 = 153,000$ However, not all combinations are generated because, after a fixed number of attempts to get a valid input-output pair, it iterates to the next rule.

2. **Object Modification** and **Property Detection**: These two are simplified generators from the Conditional Transform. In Object Modification, the 50 transformations are applied to the objects without checking any conditions. For Property Detection, a condition is checked, and the object is either kept or excluded from the output grid based on whether it meets the condition. Note that these are still different rules than the Conditional Transform.

3. **Grid Level**: For this schema, the rule is applied to the whole input grid, not to a specific object. Here, 102 different transformations are defined at the grid level, and these are then combined so that two transformations are applied sequentially to the grid for example mirror and rotate the grid. In Figure 3.1, C provides an example.

4. **Order Relation**: This is again done at the object level. The task is to find an order rule, such as ranking objects from small to big. An example can be found in Figure 3.1, D.

If one were to generate input-output pairs with a configuration of 10 maximum tries, the resulting number of tasks would be 13,752 for "Conditional Transform," 15483 for "Property Detection", 3136 for "Object Modification" 782 for "Order Relations," and 7,537 for "Grid Level." Since the height, width, object shape, color, and number of objects are randomly initialized, each task results in a unique task. This allows for looping the generation process while maintaining individual variability in each task, even though the rules themselves are repeated.

The second data generator method which was also designed by Micheal Holder is the **reverse engineering**. Creating new examples for any ARC task using

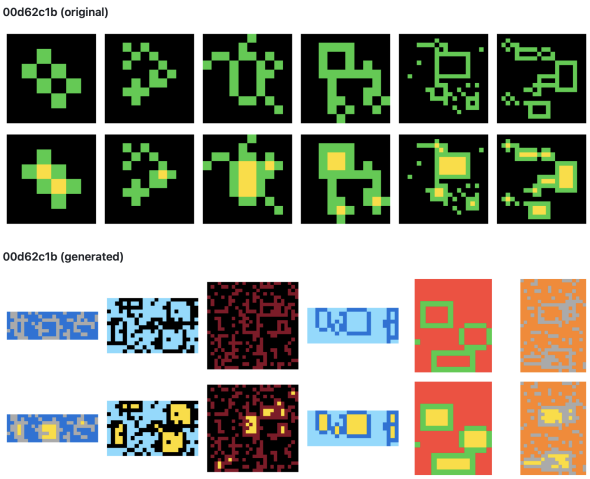


Figure 3.2: The Original ARC task (upper) and the generated one (bottom)

fixed augmentation methods results in minimal variety and can often break the examples, making this approach generally ineffective. Instead, the idea of reverse engineering is that new examples should be built from scratch. Each independent aspect, such as grid dimensions or the number of objects, is randomly determined for each example. This includes every two components that do not logically depend upon each other sampling integers for grid dimensions, numbers of symbols or objects, generating randomly shaped and colored objects, and randomly placing objects on a grid. For each of the 400 tasks, there is a generator, which is an independent Python function that solely uses the domain-specific language (DSL) and functions from Python's standard library's random module. In Figure 3.2 is an example of the Original ARC task and the generated one.

Here is a simplified explanation of Object Generation and Placement: To create an object, first, one decides its size in pixels. Then, pixels are added next to each other to form the object, ensuring it doesn't exceed certain height or width limits. Finally, the object is colored. There are two ways to place objects on a grid:

- Rejection Sampling: One picks a spot and checks if the object fits without overlapping others. If it doesn't fit, another spot is tried.
- Pre-filtered Locations: A list of spots that are guaranteed to work is used to avoid too many failed attempts.

Instead of worrying about the object's orientation (such as rotation or flipping) during creation, one creates the object normally. Later, the entire grid, including the object, is rotated or flipped to achieve the desired orientation. For some tasks, it is easier to start with the final result and work backward. Instead of

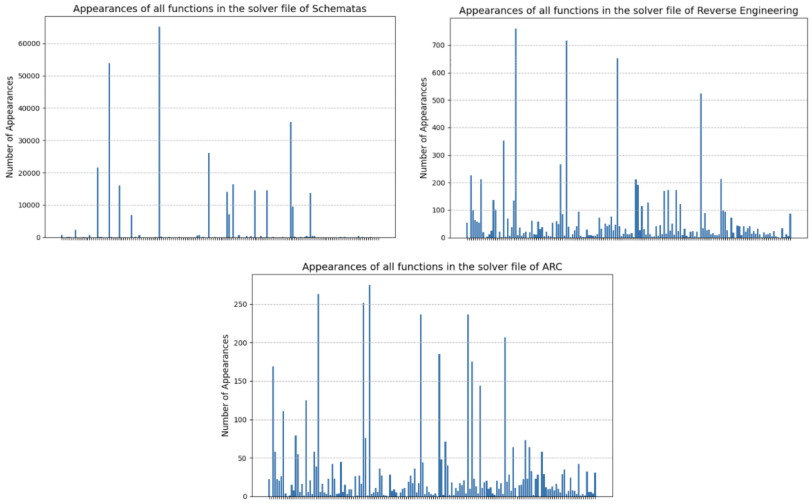


Figure 3.3: Distribution of 160 DSL functions in the solver's file. This analysis includes 22k solvers for the Schematas and 400 solvers for both the Real ARC solver and the Reverse Engineered solver. The x-axis represents the 160 discrete points of the function (names are omitted for readability), and the y-axis shows the number of occurrences in all solvers.

transforming an input grid to get the output, one creates the output grid first and then applies the reverse process to get the input grid. This approach is used when the reverse process is simpler to implement. After the input-output pair is generated the verifier checks the task for validity by passing the input grid through the solver and comparing it with the generated output grid. If they are equal the task is valid. Each verifier also works on all the original examples of the corresponding tasks in the ARC dataset. Micheal also introduces a difficulty score which refers to how challenging a task is. More elements to handle (like rows, pixels, symbols, objects) usually make an example more difficult.

In conclusion, we have two generators: **Reverse Engineering** and **Schematas**. Both provide an infinite number of tasks but only a finite number of applied rules between each input-output pair. The ideal generator would offer an infinite number of rules for training. It is debatable whether a deep learning approach can generate out-of-distribution rules, as this is difficult to measure. The goal, however, is to provide sufficient data for the machine to learn the core prior knowledge and the DSL, rather than preparing the machine for every possible rule.

To achieve this, we must examine the distribution of DSL functions used in the solvers for the generated tasks. During the training process for the language model, the grid pairs serve as the input, and the generated solver acts as the ground truth target. While humans can read and apply a DSL function definition directly, the language model needs to encounter the DSL function used as a target in various contexts to learn it effectively. In Figure 3.3, one can see the

distribution of DSL functions (a total of 160) in the solvers for Reverse Engineering, Schematas, and the real ARC solver. It is evident that the Schematas miss out on many DSL functions, with only 51 out of 160 being used. This number would not increase by looping the generation process because the solver rules and with it the DSL solver functions are fixed, and only the grid is randomly sampled. Thus, this dataset alone would not lead to a good understanding of the DSL. For Reverse Engineering and ARC, the distributions are similar, which is expected since they share the same solver function. The distribution is not identical because Michael hand-improved it over time, and the ARC solver is older than Reverse Engineering. Both use 151 of the 160 functions. Training on Reverse Engineering and testing on the real ARC training dataset would not make a great metric since they use the same rule for solving the input/output pairs. This would likely lead to a local generalization challenge (as defined in subsection 1.1.1), which is not the goal.

In the end, I think both datasets provide a solid foundation for training. The Reverse Engineering dataset represents the solvers very well, similar to the real training dataset, while the Schematas introduce new types of tasks with their own rules.

Chapter 4

Classification with Stormer

In the last chapter, we introduced a Domain-Specific Language (DSL) and the generation of data. Now, we will use this to train an encoder transformer model for a classification task. Our goal is to test its functionality before enhancing it to a language model with an autoregressive encoder-decoder transformer architecture. We used the "Stormer" architecture as a baseline from Yassine's Master Thesis. In his thesis, the Stormer architecture was used to directly generate the output grid with the Decoder. To provide a clear understanding, this chapter will introduce the theory behind language models and the famous transformer architecture. It's important to note that in this chapter, we will not implement a classical language model. Instead, we will use an encoder for multilabel classification and plan to utilize a pre-trained large language model in the next chapter. But the understanding the concept of a language model is essential to grasping the idea behind the transformer architecture.

4.1 Language Models

This definition is based on the Lecture Notes of Ryan Cotterell [6], who teaches the course Large Language Models at ETH Zurich. Given an alphabet Σ and a distinguished end-of-sequence symbol EOS $\notin \Sigma$, a language model is a collection of conditional probability distributions $p(y \mid \mathbf{y})$ for $y \in \Sigma \cup \{\text{EOS}\}$ and $\mathbf{y} \in \Sigma^*$, the Kleene closure of Σ. $p(y \mid \mathbf{y})$ therefore represents the probability of y being the next token given the history $\mathbf{y}$. The definition of a language model that is implicitly assumed in most papers on language modeling can be simply written down as the following autoregressive factorization

$$p(\mathbf{y}) = p(y_1 \dots y_T) = p(\text{EOS} \mid \mathbf{y}) \prod_{t=1}^{T} p(y_t \mid \mathbf{y}_{<t}) \tag{2.1}$$

Let's clarify the definition a bit. The alphabet Σ contains all the symbols we want to generate in a text. For example, for an English-written book, Σ would contain the entire alphabet (a, A, b, B, c, C, ...) and a few extra symbols such as the

dot, comma, semicolon, whitespace, and so on. The Kleene star $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ contains all possible finite strings. Hence, the string $\mathbf{y} \in \Sigma^*$ can be, for example, a word, a sentence, a whole article, and so on. Therefore, $p(y \mid \mathbf{y})$ gives us the probability of the next symbol y given a word, sentence, etc., $\mathbf{y}$. For example, $p(\text{g} \mid \text{do})$ should be higher than $p(\text{o} \mid \text{do})$. The end-of-sequence symbol, as its name suggests, indicates the end of a string, text, or sentence; once it is reached, there will be no further generation.

In practice, the alphabet Σ is the set of tokens, and the probability $p(y_t \mid \mathbf{y}_{<t})$ is modeled using deep learning algorithms. The process of text generation can be described as follows: Starting with an initial string $\mathbf{y}_0$, one iteratively samples the next symbol y_t from the probability distribution $p(y_t \mid \mathbf{y}_{<t})$. This continues until the end-of-sequence symbol is generated or a predefined length is reached. The choice of deep learning model significantly affects the quality of the generated text. Commonly used models include recurrent neural networks (RNNs), long short-term memory networks (LSTMs), and more recently, transformers.

4.2 Transformer

In this section, we will briefly introduce the transformer architecture from "Attention is all you Need" [7] to describe the standard architectural components of a Language Model. As already explained above there are several steps for the language model, here is a brief overview and then a more detailed description:

1. **Tokenize Words**: Split the input text into tokens.
2. **Embed Tokens**: Convert tokens into vector embeddings.
3. **Add Positional Encoding**: Add positional information to embeddings.
4. **Self-Attention Layer**: Capture contextual dependencies between tokens.
5. **Multi-Headed Self-Attention**: Use multiple attention heads to focus on different parts of the input.
6. **Feed-Forward Network**: Process through a fully connected network.
7. **Probability Score**: Generate a vector with probability scores for each token.
8. **Softmax Layer**: Normalize scores into probabilities.
9. **Predict Output**: Select the token with the highest score as the predicted output.

Tokenize Words In natural language processing, tokenization refers to converting text into a sequence of tokens. There are three main methods for this: character-level tokenization, word-level tokenization, and subword tokenization. Subword tokenization, the most commonly used method today, divides text into common word segments and single characters, balancing sequence length and vocabulary size. The final vocabulary includes special tokens for the beginning and end of sequences and for masked language modeling. Text is ultimately represented as a sequence of token IDs, corresponding to its (sub)words, and is preceded by a beginning-of-sequence token and followed by an end-of-sequence token. [8]

Embed Tokens The token embedding learns to represent each vocabulary element as a vector in $e \in \mathbb{R}^{d_e}$ from the embedding matrix $e = W_e[:, v]$, where $v \in \{1, 2, \ldots, N\}$ is the tokens id and $W_e \in \mathbb{R}^{d_e \times N}$. [8]

Add Positional Encoding The positional embedding learns to represent a token's position in a sequence as a vector in $\mathbb{R}^{d_e}$. The purpose of the positional embedding is to allow a Transformer to make sense of word ordering. Learned positional embeddings require that the input sequence length be at most some fixed number l_{max}, as the size of the learned positional embedding matrix must be finite and fixed in advance of training.

$$W_p[2i-1, t] = \sin\left(\frac{t}{\ell_{\max}^{2i/d_e}}\right), \tag{4.1}$$

$$W_p[2i, t] = \cos\left(\frac{t}{\ell_{\max}^{2i/d_e}}\right) \tag{4.2}$$

$$e = W_e[:, x[t]] + W_p[:, t] \tag{4.3}$$

The original Transformer model described in [7] uses hard-coded positional embeddings, where the Positional Encoding is not learned and $l_{max} = 10,000$

Self-Attention Layer Attention is the key component of transformer architectures. It is a mechanism that enables the model to concentrate on various parts of the input sequence while processing a specific element. By assigning weights to each input element according to its relevance to the current element, the model can dynamically adjust the importance of different elements. This allows the model to capture long-range dependencies in sequences, which is particularly useful for tasks such as natural language understanding and generation.

The Attention layer is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.4}$$

$$\text{with } \text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n}\exp(x_j)} \tag{4.5}$$

The Q is the quarry, the K is called the key and the V is the value:

$$Q = W_Q E \quad \text{with } W_Q \in \mathbb{R}^{d_{model} \times d_e} \tag{4.6}$$

$$K = W_K E \quad \text{with } W_K \in \mathbb{R}^{d_{model} \times d_e} \tag{4.7}$$

$$V = W_V E \quad \text{with } W_V \in \mathbb{R}^{d_{out} \times d_e} \tag{4.8}$$

Note that E is the concatenated sequence of our input tokens $E = [e_1, e_2, \ldots, e_N]$ and in the paper [7] they also add a bias term. The weights W are learned.

Multi-Head Self-Attention : Instead of using a single attention function, the model employs multiple attention mechanisms, called "heads," in parallel. Each head independently computes the attention function and produces an output. The outputs of all heads are then concatenated and linearly transformed to form the final output. This allows the model to focus on different parts of the input sequence from multiple perspectives.

$$\text{MultiHead}(E) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{4.9}$$

$$\text{head}_i = \text{Attention}(W_Q^i E, W_K^i E, W_V^i E) \tag{4.10}$$

Feed-Forward Network Following the multi-head self-attention sub-layer, a fully connected feedforward network is applied to each position separately and identically. This network consists of two linear transformations with a ReLU activation in between. To ensure stable training and to help the model generalize better, both sub-layers in each layer are followed by residual connections and layer normalization. The residual connection adds the input of the sub-layer to its output, and layer normalization normalizes the summed output.

At the final layer of the network, an unembedding step is performed where the high-dimensional output vectors are projected back to the vocabulary size using a linear transformation. This is followed by a softmax activation function, which converts the logits into probabilities for each possible output token. The token with the highest probability is then selected as the predicted output.

Encoder vs. Decoder The architecture involves two main stages: encoding the context sequence with bidirectional multi-head attention, resulting in vector representations of each context token, and then encoding the primary sequence,

where each token can use information from both the encoded context and preceding primary tokens. Encoder-only transformers like BERT are trained on masked language modeling to learn useful text representations for various NLP tasks. In contrast, decoder-only transformers like GPT use unidirectional attention for autoregressive language modeling to predict the next token in a sequence, differing from BERT in their attention mechanism and layer-norm application. It is important to note that the limitation of the transformer lies in the computational complexity of matrix multiplication in Equation 4.4, which is $\mathcal{O}(n^2 \cdot d)$. Here, n represents the sequence length and d denotes the representation dimension.

4.3 Stormer and Classification

First, a simple classification problem was implemented. This makes sense because an autoregressive transformer essentially addresses a classification problem for predicting the next most probable token. The task involved predicting which of the 160 DSL functions appear in the solver, using input-output pairs. This is a multi-label classification problem because multiple DSL functions can occur simultaneously in the solver. Consequently, a sigmoid activation function is used at the end instead of a softmax function. I will first explain the architecture we used, followed by a discussion of the results and our decision to use a pre-trained model.

The ARC challenge for each task consists of a set of k input-output pairs, with $k-1$ pairs used for "training" and 1 pair used for testing. The Stormer architecture from Yassine's Master Thesis was employed to directly generate the output grid for the k-th testing grid. The $k-1$ pairs were used as input for the encoder, and the last input grid was fed into the decoder. However, since that was not our primary objective, we encoded all the pairs and skipped the decoder part, instead using a fully connected layer for classification. Essentially, we transformed the original encoder-decoder Stormer into an encoder-only transformer architecture.

In the context of a specific ARC task, a total of $2k$ single grids are utilized, each represented as $G \in \mathbb{R}^{M \times N \times 10}$. Here, M denotes the number of rows, N denotes the number of columns, and there are 10 distinct colors represented as separate channels. To simplify the process and align with architectural decisions, all grids are padded to a fixed size of 30×30, the maximum grid size in the ARC challenge. A one-hot encoder is used for the 10 colors, and an 11th channel is added for padding, treating it as an additional color. Consequently, the grids are represented as $G \in \mathbb{R}^{30 \times 30 \times 11}$. Instead of tokenizing the task first like in the Visual Transformer [9], the task is directly embedded with convolutional neural networks (CNN) [10]. Four different CNNs with different configurations are used in parallel with depth of one, where the receptive field equals the size of the kernel. This is illustrated in Algorithm 1. The input dimension for the encoder should be $\mathbb{R}^{N \times M \times d_e}$, where $d_e = 80$. Zero-padding is applied to the output of the

Algorithm 1 Convolutional Embedding

Conv1: **in_filters** = 1l; **out_filters** = 1l; **kernel_size** = 1; **stride** = 1
Conv2: **in_filters** = 1l; **out_filters** = α; **kernel_size** = 3; **stride** = 1
Conv3: **in_filters** = 1l; **out_filters** = β; **kernel_size** = 5; **stride** = 1
Conv4: **in_filters** = 1l; **out_filters** = γ; **kernel_size** = 11; **stride** = 1

$c1 \leftarrow \text{Conv1}(G)$
$c2 \leftarrow \text{Conv2}(G)$
$c3 \leftarrow \text{Conv3}(G)$
$c4 \leftarrow \text{Conv4}(G)$

return: $G' \leftarrow \text{concat}(c1, c2, c3, c4)$

convolutions to maintain the $N \times M$ dimension for concatenation. In practice, $\alpha = 15$, $\beta = 20$, and $\gamma = 33$. An additional channel with either 0 or 1 is included to indicate whether the data is from an input or output grid. Then the 2D positional encoding is added, which is defined as:

$$PE(m, n, 2i) = \sin\left(\frac{x}{10000^{4i/d_e}}\right)$$
$$PE(m, n, 2i+1) = \cos\left(\frac{x}{10000^{4i/d_e}}\right)$$
$$PE(m, n, 2j + d_e/2) = \sin\left(\frac{y}{10000^{4j/d_e}}\right)$$
$$PE(m, n, 2j + 1 + d_e/2) = \cos\left(\frac{y}{10000^{4j/d_e}}\right)$$

Where $0 < i, j < d_e$, we obtain the positional encoding matrix $e_p \in \mathbb{R}^{M \times N \times d_e}$. The complexity of transformers scales quadratically with sequence length, making self-attention operations $O(n^2 \cdot d)$. To handle large grids on available GPUs, the sequence length must be reduced by passing the k grid pairs separately through a single trainable encoder. This process is illustrated in Figure 4.1, where (A) shows the preprocessing of a single input-output grid, and (B) depicts the process for all pairs. The memory in Figure 4.1 (B) was used to pass through the decoder for the original one. However, in this approach, a fully connected layer is first used to map the stacked dimension of $K \cdot d_e$ channels of the memory to d_e channels. Additionally, max pooling and average pooling layers are employed to scale down the flattened $N \times M$ to two values each. These layers are concatenated, resulting in a shape of $4 \times d_e$. Finally, a single fully connected layer with a sigmoid activation function is used to map the output to 160 values of DSL. If a value exceeds 0.5, the model assumes that the DSL function is part of the Solver function.

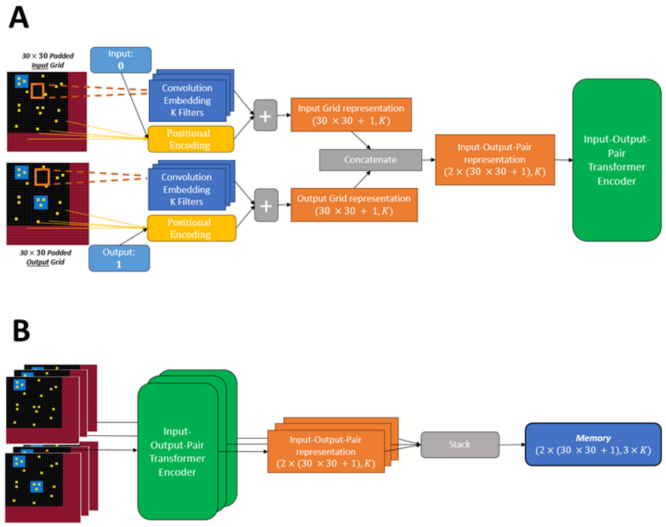


Figure 4.1: architecture overview of the Stormer. A) Encoding pipeline for a single Input-Output pair. B) Encoding pipeline for the entirety of the training set of Input-Output pairs. This figure is from the Master Thesis from Yassine

4.3.1 Results

For training, we used tasks generated by the **Schematas** Generator, and for evaluation, a smaller set of tasks from the same generator. The training dataset uses 20,000 tasks from object modifications, 96,000 tasks from property detection, 87,000 tasks from conditional transform, 8,000 from grid level, and 6,000 from object relation. The encoder was initialized with 8 layers and 4 heads per layer, resulting in a total of 1,113,740 parameters. We used precision and the number of forgotten functions as metrics. The number of forgotten functions, equivalent to the number of false positives, represents functions that should have been predicted but were not. Because false positives are more detrimental than unnecessary additional predicted functions. The precision is calculated as the number of correctly predicted labels divided by 160. Figure 4.2 shows the results of different Schematas Conditional Transform, Grid Level, and Order Relation along with the loss function. It is evident that the loss converges quickly, but the metric does not improve, indicating potential overfitting. An example output is provided in the appendix section A.2. Based on these results, we decided to use pre-trained models. Pre-trained large language models (LLMs) are often preferred over training from scratch for generalization tasks due to their ability to leverage knowledge acquired during pre-training on massive datasets. Initially, Stormer was used for generating output grids directly, but now we aim for a sequence of tokens as output, similar to LLMs. Thus, we chose to proceed with the T5 model.

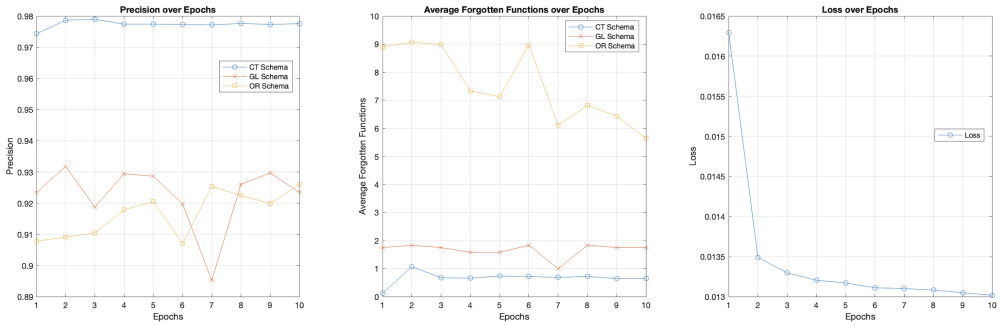


Figure 4.2: Figure 1: Comparison of performance metrics over 10 epochs for three schema types (CT, GL, and OR). The left subplot displays precision, the middle subplot shows average forgotten functions and the right subplot illustrates loss. CT Schema is represented with circles, GL Schema with crosses, and OR Schema with squares.

Chapter 5

T5: Task-to-Solver

In the last chapter, we mentioned our decision to use a pre-trained Large Language Model to generate the DSL solver for the ARC dataset. In this chapter, I will focus on the core part of this thesis: introducing the "Text-to-Text Transfer Transformer" (T5) and detailing my implementation. This involves representing the ARC tasks and the DSL solver functions as T5 tokens. Additionally, I will discuss the hyperparameters, metrics, optimizer, and loss functions used for training. Furthermore, I will examine the results, the challenges encountered, and the proposed solutions to address these challenges.

5.1 Text-to-Text Transfer Transformer

In this section, I will introduce the T5 model from the paper "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer" [11]. Known as the “Text-to-Text Transfer Transformer” (T5), this model utilizes a standard transformer architecture. Unlike modern transfer learning approaches that often employ a single Transformer stack for tasks like language modeling or classification, T5 uses an encoder-decoder structure. The encoder consists of a stack of blocks, each comprising two subcomponents: a self-attention layer followed by a small feed-forward network. Layer normalization is applied to the input of each subcomponent. Both the encoder and decoder consist of 12 blocks. The feed-forward networks include a dense layer with an output dimensionality of 3072, a ReLU nonlinearity, and another dense layer. Attention mechanisms use "key" and "value" matrices with an inner dimensionality of 64 and have 12 heads. All other sub-layers and embeddings have a dimensionality of 768. The T5 model has about 220 million parameters, roughly twice that of a BERT model. Dropout regularization is applied throughout the model with a probability of 0.1. Unlike the original Transformer, which uses sinusoidal position signals or learned position embeddings, the T5 model employs relative position embeddings. These embeddings depend on the relative distance between elements, not fixed positions. Specifically, the embedding is based on the offset between the "key" and "query" in self-attention. In this approach, each position embedding is a scalar

added to the logits for computing attention weights. To enhance efficiency, the same position embedding parameters are shared across all layers, but different attention heads within a layer use different learned position embeddings. This is based on the paper "Self-Attention with Relative Position Representations" [12]:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V) \tag{5.1}$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \tag{5.2}$$

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}} \tag{5.3}$$

$$a_{ij}^K = w_{\text{clip}(j-i,k)}^K \tag{5.4}$$

$$a_{ij}^V = w_{\text{clip}(j-i,k)}^V \tag{5.5}$$

$$\text{clip}(x,k) = \max(-k, \min(k,x)) \tag{5.6}$$

From Equation 5.1 to Equation 5.3 the Attention layer is demonstrated with the added offset for the relative position representations. The relative position representations $w^K = (w_1^K, \ldots, w_k^K)$ and $w^V = (w_1^V, \ldots, w_k^V)$ are learned in Equation 5.4 and Equation 5.5. They use clipping (Equation 5.6) because it helps to constrain the relative position to a manageable range, ensuring that the model does not need to learn an unbounded number of position embeddings. This clipping function restricts the relative position values to be within the range of $[-k, k]$, where k is a predefined constant. By doing so, it simplifies the learning process and enhances the model's ability to generalize across different sequence lengths. With that, the T5 allows for a more flexible and efficient handling of positional information.

The T5 uses SentencePiece to encode text [13] as WordPiece tokens [14]. The T5 has a vocabulary of 32,000 wordpieces/tokens.

For training the T5 they introduced "Colossal Clean Crawled Corpus" (C4), a data set consisting of hundreds of gigabytes of clean English text scraped from the web. A denoising objective inspired by BERT's masked language modeling was used. This involved randomly dropping out 15% of tokens from the input sequence and training the model to predict these dropped-out tokens using unique sentinel tokens. The model was fine-tuned for 262,144 steps on each downstream task using a constant learning rate of 0.001. A fine-tuning phase was performed after this pre-training phase to adapt the model to specific tasks (e.g. translation of English to German). The AdaFactor optimizer and the Learning Rate Schedule were used for both pre-training and fine-tuning.

The T5 is divided into different models, which come with different numbers of parameters, this can be seen in Table 5.1.

Ending this brief summary of the T5, one must say that, implementing a pre-

Model	Parameters	# layers	d_{model}	d_{ff}	d_{kv}	# heads
Small	60M	6	512	2048	64	8
Base	220M	12	768	3072	64	12
Large	770M	24	1024	4096	64	16
3B (XL)	3B	24	1024	16384	128	32
11B (XXL)	11B	24	1024	65536	128	128

Table 5.1: Model Specifications

trained Large Language Model like T5 for the DSL and ARC tasks presents significant challenges. The T5 is designed as a sequence-to-sequence model, whereas our data resembles an image-to-sequence format. This distinction is crucial because generating Python code demands high precision; even a single error renders the DSL solver unusable. Unlike natural language, which allows multiple ways to convey the same message, our output must be highly robust to ensure accuracy and consistency.

5.2 Representation and Tokenization

In this section, I will present the representation and tokenization process I applied to the tasks and DSLs to prepare them for input into the T5 model. Unfortunately, we cannot use the same embedding method as we did for the Stormer, because we don't have access to the embeddings in T5. Instead, we need to map the tasks to T5 tokens, after which the pre-trained and inaccessible embeddings can be utilized.

First, I will discuss the tokenization of the DSL solver and then the tasks.

5.2.1 Solver

The goal is to create a sparse and robust representation of the DSL solver. For that, we need to eliminate unnecessary Python syntax. This reduces the risk of errors and ensures the solver function initializes correctly. By doing so, we can fully leverage the solver's format. Here's the example illustrated in item 3.1:

```
def solve_00d62c1b(I):
    x1 = objects(I, T, F, F)
    x2 = colorfilter(x1, ZERO)
    x3 = rbind(bordering, I)
    x4 = compose(flip, x3)
    x5 = mfilter(x2, x4)
    O = fill(I, FOUR, x5)
    return O
```

The T5 tokenizer would convert this string into 94 tokens, as shown in Appendix section A.3. Note that '_' is not an underscore but the '\u581' character, indicating a free space. Despite this, the solver's format remains straightforward: assigned variable, DSL function, and arguments, each followed by a new line. We can optimize this format.

First, we can omit '`def solve_00d62c1b(I):`' since the input grid I is always our function's input, and the specific task name '`00d62c1b`' is irrelevant to the LLM. Then, we can remove all Python syntax symbols like brackets, commas, spaces, equal signs, and the return line. To separate the functions and its arguments I use the simple ";". And of course to signal a newline I use the "#newline". This leads to a representation:

```
#BoF
#newline x1 objects ; I T F F
#newline x2 colorfilter ; x1 ZERO
#newline x3 rbind ; bordering I
#newline x4 compose ; flip x3
#newline x5 mfilter ; x2 x4
#newline O fill ; I FOUR x5
#EoF
```

The symbols "#BoF" and "#EoF" stand for "Begin of Function" and "End of Function." From this, the code is perfectly reconstructable. Tokenizing these strings directly often leads to suboptimal results because, for example, "#BoF" is split into four different tokens instead of being treated as a single token. This would diminish its utility.

To address this, we first convert this representation into a list, ensuring there are no line breaks or spaces. Then, we map all the entries of that list into tokens of the T5 model. The tokenization mapping to the T5 model is done before training and is straightforward since the number of symbols/words to map is constant, except for the integer i in `"x" + str(i)`. I have never encountered a solver with an index higher than `x200`. To ensure completeness, I extended the range from `x1` to `x350`. With that, there are approximately 500 symbols to map, including constants of the DSL (e.g., `ZERO`, `FOUR`), the 160 DSL functions, and a few symbols introduced earlier (e.g. "#BoF", "#EoF", ";", ...).

The mapping is carried out in three stages:

1. **Fixed mappings**: Each `"x" + str(i)` is mapped to a capital letter in the alphabet. After reaching `"Z"`, the sequence continues with `"AA"`, `"AB"`, and so on. If a capital letter combination is not available in the embedding, it is skipped (e.g., `BF`). The "#EoF" symbol is mapped to the End-of-Sequence token by the T5 "</s>".

2. **Direct Matching with T5 Tokens**: For the remaining symbols, a direct match is sought within the T5 token set. Words like `objects` and `fill` can be mapped directly to their corresponding T5 tokens.

3. **Levenshtein Distance for Similar Tokens**: If no perfect match is found, the Levenshtein distance is used to find the closest similar token. This mapping is an injective function, ensuring that no two words map to the same token.

Of the 493 words to map, 350 are mapped using the first method, 41 using the second, and 102 using the Levenshtein distance.

Additionally, a slightly different approach was implemented for the representation where the mapping of `x1` is separated into `x` and `1`, as follows:

```
#BoF
#newline x 1 objects ; I T F F
#newline x 2 colorfilter ; x 1 ZERO
#newline x 3 rbind ; bordering I
#newline x 4 compose ; flip x3
#newline x 5 mfilter ; x 2 x 4
#newline O fill ; I FOUR x 5
#EoF
```

With this approach, we no longer need to worry about how high the integer `i` gets. We simply map `x` to the T5 token `y`, as `x` is already used for tokenizing the Task Grids. The numbers are mapped to their corresponding values, already represented in the T5 tokens. This reduces the number of used tokens to 205. For simplicity, we will call this method **Separating Variables**, and the first method **Variable to Alphabet**.

Next, I will introduce the representational tokenization of the task grid. It is important to note that the union of the T5 tokens used for the task grid and the solver is an empty set.

5.2.2 Task Grid

Unfortunately, in the case of a pre-trained model, we do not have access to the encoder and decoder, making it impossible to pass input-output pairs sequentially through the encoder, as we did for the Stormer. For example, a task typically has four input-output pairs, resulting in eight grids, each being a matrix with a maximum height and width of 30, containing entries from zero to nine. If we naively input this into a transformer, it would result in a context length of $30^2 \times 8 = 7,200$, which is too high for our GPUs to handle. Therefore, we must devise a way to represent the matrices to pass them into the T5 model.

Fortunately, the T5 model can be initialized with any context length, as its positional encoding is computed differently than the original transformer. This means the only limitation is the computational power of our GPUs.

We decided that the task representation should be fully reconstructable, ensuring no loss of information. This and the T5 model's limited token capacity for mapping windows rules out the Visual Transformer approach described in "An Image is Worth 16x16 Words" paper. [9]

For representation, all numbers were mapped to the single token colors, resulting in a dictionary as follows:

```
dic = {0: '_Black', 1: '_Blue', 2: '_Red', 3: '_Green', 4: '_Yellow',
       5: '_Gray', 6: '_Purple', 7: '_Orange', 8: '_Azure',
       9: '_Brown', 10: '_White'}
```

The rows were iteratively examined, and any color repeated more than three times was converted into a count-token format ({number of times}x{Color}). For example, if a row consists entirely of 0s, it would be represented by the three tokens $[30, x, \_Zero]$, thereby reducing 30 tokens to three. Additionally, the terms "input" and "output" signal an input and an output grid, and a new row is indicated with a ';' symbol. An example tokenized grid can be found in the appendix section A.3. In Figure 5.1, one can see the distribution of the number of tasks over the context length. For the Schematas generator, the height and width are sampled with a normal distribution, resulting in the context length distribution shown in the figure. This entire dataset leads to an average context length of 1679.54.

In summary, the average context length of our tokenized and represented real dataset varies across the training (789.8), evaluation (1284.24), and testing (1406.41) sets. We ultimately choose a context length of $l_x = 1024$, which accommodates the majority of grids. In cases where the context length surpasses l_x, the sequence is truncated up to the last complete input-output pair.

To conclude, we have detailed our task and DSL solver representation and tokenization process. The tokenized input-output grids serve as the input sequence for T5, while the DSL solver tokens constitute the target output sequence.

5.3 Implementation

In this section, I will explain the implementation used to fine-tune the T5 model on the ARC Challenge. Throughout the thesis, I have already explained the inference pipeline of the large language model, but let me now provide more detailed steps:

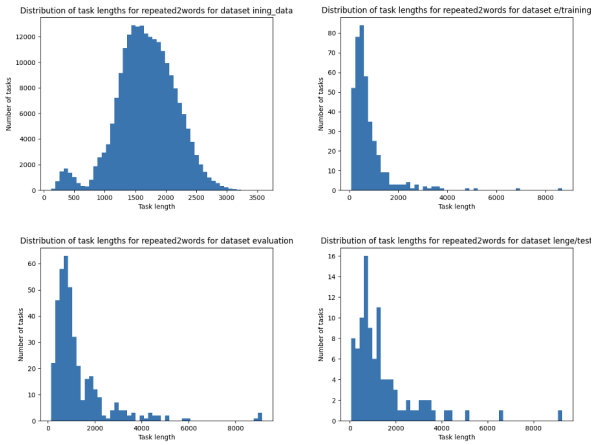


Figure 5.1: In this figure, the y-axis represents the number of tasks, and the x-axis represents the context length of the tasks after applying the representation. The top left corner shows a total of 218k tasks for the Generator Schematas. The top right corner displays the distribution of the training data from the real ARC Dataset. The bottom left corner illustrates the Evaluation ARC Dataset, while the bottom right corner represents the test dataset, with one grid missing since it is a hidden dataset and the last output grid is not given.

1. For our dataset, we have a given task t_i with different numbers of input and output grids, represented as $t_i = \{(I_1, O_1), (I_2, O_2), \ldots, (I_N, O_N)\}_i$. Each pair of grids follows a specific rule r_i. However, rules can be shared across tasks, such that $t_i \neq t_j$ with $r_i = r_j$ for $i \neq j$ (especially in the case of generated data).

2. For each task t_i, we have a DSL solver function ρ_i, where $\rho_i(I_n) = O_n \; \forall n \in \{1, \ldots, N\}_i$.

3. The task and the solver are represented and tokenized (as described in the previous section) to obtain ρ_i^M and t_i^M.

4. We use the model M (T5) to predict $\hat{\rho}_i^M$, such that $M(t_i^M) = \hat{\rho}_i^M$.

5. The generated $\hat{\rho}_i^M$ is then mapped back to the original form with a simple Python function: $\text{mapback}(\hat{\rho}_i^M) = \hat{\rho}_i$.

6. If $(\hat{\rho}_i(I_n) = \hat{O}_n) == O_n \; \forall n \in \{1, \ldots, N\}$, then the generation was successful. To demonstrate that the model understands the DSL and can generalize it, the results should satisfy $(\hat{\rho}_i(I_n) = \hat{O}_n) == O_n \; \forall n \in \{1, \ldots, N\}$ with $\hat{\rho}_i \neq \rho_i$.

5.3.1 Training

The T5 model is trained on tasks from two sets: **Schematas** and **Reverse Engineering**. The Schematas set contains 218,000 tasks, the same as those used for the Stormer model. The Reverse Engineering set includes 23,000 tasks, with generation evenly distributed across all original ARC tasks. Both the input and target sequence context lengths are set to 1048. Truncation in the target sequence is more critical than in the input because the entire solution needs to be generated correctly to solve the task, whereas omitting one of for example the five grid pairs in the input is less problematic. The '__getitem__' method of the DataLoader includes a warning print statement indicating when truncation occurs in the target sequence.

The Adam optimizer [15] is used for backpropagation, and the mean Cross Entropy loss is calculated between each token in the predicted sequence $\hat{\rho}^M$ and the ground truth sequence ρ^M.

5.3.2 Testing

For testing, we use the Real ARC dataset and the written DSL solvers ρ as ground truth. Additionally, we include extra generated samples from different schemata: Conditional Transform, Grid Level, and Order Relation. These are generated using a DSL solver function, which serves as the ground truth in these cases. As metrics, we use:

- Average BLEU Score
- Levenshtein Distance
- Reconstruction accuracy (if the output can be reconstructed to a solver function)
- Generation capability (if the solver function can generate an output)
- Overall accuracy

BLEU Score The BLEU (Bilingual Evaluation Understudy) score is calculated by comparing a candidate sequence $\hat{y}$ with a list of reference sequence $(y^{(1)}, \ldots, y^{(N)})$.

1. **N-gram Precision:** Given any sequence of tokens $y = y_1 y_2 \ldots y_K$, and any integer $n \geq 1$, we define the set of its n-grams to be:

$$G_n(y) = \{y_1 \cdots y_n, y_2 \cdots y_{n+1}, \ldots, y_{K-n+1} \cdots y_K\}$$

2. **Modified N-gram Precision:** Define the modified n-gram precision function as:
$$p_n(\hat{y}; y) := \frac{\sum_{s \in G_n(\hat{y})} \min(C(s, \hat{y}), C(s, y))}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})}$$
where $C(s, y)$ is the number of appearances of s as a sub-sequence of y.

3. **Geometric Mean:** The geometric mean of the modified n-gram previsions up to length N is:
$$\left(\prod_{n=1}^{N} p_n \right)^{\frac{1}{N}}$$

4. **Brevity Penalty:** To penalize for shorter candidate sequences, one include a brevity penalty BP, defined as:
$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-\frac{r}{c}} & \text{if } c \leq r \end{cases}$$
where c is the length of the candidate sequence and r is the reference sequence length.

5. **Final BLEU Score:** The BLEU score is then given by:
$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n \right)$$
where w_n are the weights (typically uniform, i.e., $w_n = \frac{1}{N}$).

In our case, $\hat{y} = \hat{\rho}^M$ and $y = \rho^M$, with $N = 4$, so we only consider up to 4-grams. The formula can be challenging to interpret. According to [16], for a translation task, the BLEU score can be interpreted as follows:

- < 10: Almost useless
- 10 - 19: Hard to get the gist
- 20 - 29: The gist is clear, but has significant grammatical errors
- 30 - 40: Understandable to good translations
- 40 - 50: High quality translations
- 50 - 60: Very high quality, adequate, and fluent translations

Levenshtein Distance The Levenshtein distance between two strings a and b, of length $|a|$ and $|b|$ respectively, is given by $\text{lev}(a, b)$, where:

$$\text{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0 \\ |b| & \text{if } |a| = 0 \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if head}(a) = \text{head}(b) \\ 1 + \min \begin{pmatrix} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \\ \text{lev}(\text{tail}(a), \text{tail}(b)) \end{pmatrix} & \text{otherwise} \end{cases}$$

The $\text{tail}(x)$ of a string x refers to the string containing all tokens of x except the first one. The $\text{head}(x)$ is the first token of x. Note that the sequences of tokens ρ^M and $\hat{\rho}^M$ are converted into strings and then compared at the character level, rather than at the token level.

Reconstruction Accuracy Reconstruction accuracy measures whether the mapping back was successful or resulted in an error, expressed as $\text{mapback}(\hat{\rho}_i^M) = \hat{\rho}_i$

Generation Capability Generation capability is indicated as one if at least one output could be obtained with .

$$\hat{\rho}_i(I_n) = \hat{O}_n \text{ for } n = \{1, \ldots, N\} \tag{5.7}$$

Accuracy Accuracy is calculated by determining if the generated output was correct or not:

$$\frac{1}{N} \sum_{i=0}^{N} \mathbf{1}(\hat{O}_i == O_i) \tag{5.8}$$

Here, **1** equals one if the statement is correct and zero otherwise.

5.4 Results

After extensive coding and debugging, the GitHub repository is now ready for training and generating models. We train on a dataset containing approximately 241,000 samples, consisting of 218,000 from the Schematas Generator and 23,000 from the Reverse Engineering. For testing, we use datasets with 782 tasks for order relations, 1,000 tasks for conditional transform, 1,000 tasks for grid level, and 400 tasks from the original ARC training dataset.

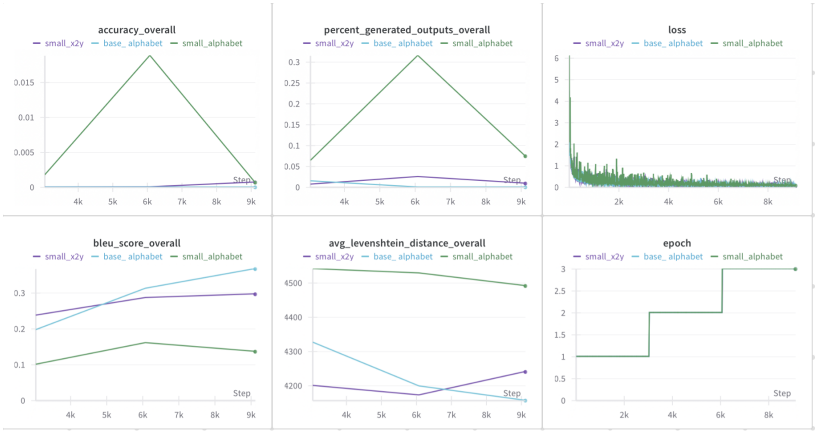


Figure 5.2: This figure presents the results for three different models: two trained on the small T5 with 60M parameters and one on the Base T5 with 220M parameters. The tokenization method "Separating Variables" is labeled as 'x2y,' while the method "Variable to Alphabet" is labeled as 'alphabet.' Training was conducted over the entire dataset for three epochs, with metrics scores obtained after each epoch. The metrics displayed include accuracy, the percentage of if the solvers are generating an output grid, the average BLEU score, and the average Levenshtein distance, all measured across four different test sets. Additionally, the right column of the figure shows the loss function over every 10th step.

The first results can be seen in Figure 5.2. The T5 small models were trained on a single Quadro RTX 6000 GPU, while the Base T5 required four of these GPUs. The models were trained for three epochs over the entire training dataset with a batch size of 8, resulting in approximately 30,125 update steps per epoch and an 8-hour training time per epoch.

The most important metrics are the accuracy and the percentage of solvers generating an output grid. There is a high correlation between these metrics because for an accuracy higher than zero, an output must be generated. Interestingly, a higher BLEU score does not directly lead to higher accuracy, though this score increases steadily with more epochs.

Additionally, the base_alphabet and small_x2y models appear to be the most stable and consistently improving, while the small_alphabet model struggles to maintain its performance gains. The most concerning issue is that during testing, the models generate an output less than 10% of the time, which is far too low (except for a sudden peak). For practical use, this number should be at 90% before worrying about accuracy.

Examining the generated samples in detail, the problem lies in the first few tokens. It is important to note that the metric for the conversion from T5 tokens back to a solver Python function, $\text{mapback}(\hat{\rho}_i^M) = \hat{\rho}_i$, is one for all models be-

Figure 5.3: Performance metrics for various models using the small T5 and Variable to Alphabet method over 10 epochs with 1,000 steps per epoch. Note that the Accuracy was constantly zero for all of them and so excluded from the figure.

cause the Python code for reconstructing back to the DSL solver is quite robust to errors. The reconstruction iterates between the "#newline" tokens and reconstructs every line independently; if a line is not reconstructable, it is converted to a comment in Python with the error.

In the Appendix (subsection A.4.1), one can see an example generated solver for a task in the Grid Level test set from the Base T5 model with "Variable to Alphabet" in the third epoch. The BLEU score and the Levenshtein distance are printed for that specific example. The number of seen pairs is a value between zero and one, indicating that 100% of pairs were covered in the context length. For the task, it is noticeable that the first few tokens are completely off, while the rest looks good. For example, the first three tokens are $[' < pad >', ' < extra\_id\_0 >', ' ZERO']$, which makes no sense at all, and the first printed variable is 'x6'.

In conclusion, it is essential to ensure that models train with greater stability and less variance, such as avoiding sudden peaks, and to achieve a more consistent decrease in loss. Additionally, it is crucial to address why the first tokens are significantly off.

To improve training stability and prevent overstepping minimums in the loss space, a learning rate scheduler using Cosine Decay has been implemented. This scheduler starts the learning rate at its maximum value and gradually decreases it to a minimum, typically zero, following a cosine function. Previously, a constant learning rate of $lr = 0.0001$ was used, which could lead to instability. The new scheduler ensures a more stable learning process.

Additionally, the number of beams in beam search has been increased to enhance the quality of the results. Unlike greedy search, which selects the single most probable word at each step, beam search tracks multiple hypotheses or potential word sequences simultaneously. Starting with the initial context or prompt, it expands each hypothesis by generating all possible next words. The probabilities of these expanded hypotheses are calculated, and only the top num_beams hypotheses with the highest probabilities are kept for further expansion [17]. In the experiments before the number of beams where set to 2, which is more similar to a greedy search.

For the new experiments in Figure 5.3, the small T5 model and the Variable to Alphabet method were used. This method was chosen due to its previous results and its robustness in reconstruction. Additionally, the model learns that x_1, x_2, etc., represent objects to move, filter, argument, and so on. The experiments were conducted over 10 epochs, with validation after each epoch, using the same validation set as before. However, this time there were only 1,000 steps per epoch, with a greater focus on validation. We measured a baseline model with the same configurations as before and four models with scheduler learning rates and different numbers of beams: 2 (*baseline_scheduler*), 5 (*num_beams_5*), 15 (*num_beams_15*), and 25 (*num_beams_25*). It is evident that the baseline model performs much worse than the other four models. For the beam search, the number of beams does not seem to make a significant difference, but beam search effectiveness depends on the model's predictive capability. For better-performing models, comparing different beam searches might be more insightful.

To address the issue with the initial tokens, two methods were explored: weighted loss and prompt engineering.

Weighted Loss: The original loss is calculated by the mean cross-entropy loss between predicted and ground truth tokens. The modification involves a weighted loss, where the loss of every individual token is multiplied by a scalar before calculating the mean. The weights are a declining linear interpolation between 1.5 and 0.5, thus penalizing errors in the initial tokens more heavily. This weighted loss is introduced after 70% of the iterations in each epoch; beforehand, the weights are uniform.

Prompt Engineering: The second method involves appending the first three tokens of the sequence (which are consistently $[\#BoF, \#newline, x1]$) to the end of the input sequence. This approach aims to guide the model in predicting the correct subsequent tokens, similar to how text prompts are used to initiate sentences in language models.

Additionally a small trick is implemented in the reconstruction, where the first three tokens are always replaced with the sequence $[\#BoF, \#newline, x1]$, since this stays the same no matter the task.

For the upcoming experiments, we will use the small T5 model with "Variable

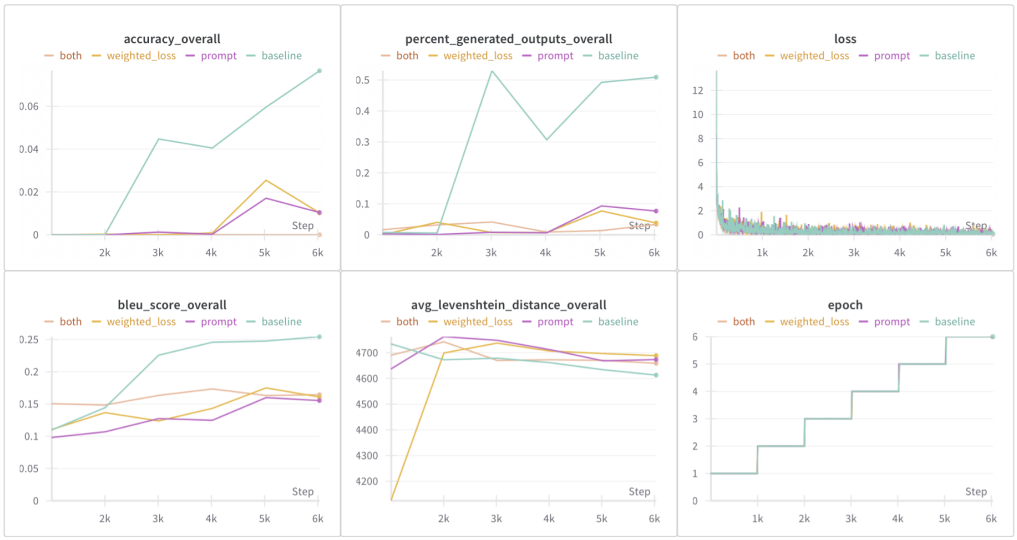


Figure 5.4: The performances of different metrics for training the small T5 over 6 Epochs and 10000 steps. The loss is measured every 10th step and it is validated after every epoch.

to Alphabet" tokenization, 15 beams, and a scheduler learning rate of 0.0001. Figure 5.4 illustrates the compression results after 6 epochs (10,000 steps each) using Prompt Engineering (prompt), Weighted Loss (weighted_loss), both (both) or neither (baseline). While both Prompt Engineering and Weighted Loss underperform compared to the baseline, they demonstrate more stable improvement in BLEU score and Levenshtein distance. Notably, these methods achieved the best results so far, generating outputs 50% of the time and reaching 7.65% accuracy on the entire test dataset. However, accuracy on the test set with real ARC training data remains at 0%. A positive finding is the generation of samples with perfect accuracy and novel solver functions compared to the ground truth shown in the appendix subsection A.4.2. This is reflected in a BLEU score of 0.24846.

Based on these results, it can be concluded that a DSL can be learned by the large language model, though there is room for improvement. A common error for the other 50% of cases, where the model failed to generate an output, was "Error solving task 2: name 'x2' is not defined." This indicates that variable 'x2' was never defined and was subsequently skipped by the solver, a small thing to

5.5 Future Work

The results presented demonstrate that solving the ARC challenge with a generated DSL solver from a large language model is feasible, though challenging. The model needs to grasp the essence of the DSL solver to achieve good results.

Unfortunately, a significant portion of this thesis was dedicated to developing the Python code and GitHub repository to establish a functional pipeline. Despite this, within just two weeks of trial and error, a 50% output generation rate and 7.65% accuracy were achieved. There are several improvements and experiments to consider for enhancing the model:

1. Every time the model generates an output grid that does not match the ground truth, it represents a new task. The generated solver encapsulates a new rule. These tasks can be utilized for subsequent training sessions, enabling the model to also serve as a task generator.

2. In the paper "On the Measure of Intelligence," the importance of a curriculum is emphasized. Rather than simply adding all generated data into the training set, one could start with simpler Schemate tasks, such as rules at the grid level, then progress to Order Relation and Conditional Transform tasks. By improving performance on these datasets first, the model can then tackle reverse engineering. Learning the DSL solver's essence is easier with Schemate tasks due to the function distribution from the DSL (see Figure 3.3).

3. The loss function could be improved. Currently, the loss is computed between ρ^M and $\hat{\rho}^M$, but it might be more beneficial to focus on the relationship between the output grids O and $\hat{O}$.

4. If results improve, new sets of tasks should be generated after each epoch, ensuring the model never encounters the same task twice. This approach could enhance the generalization process.

Other areas of research include improving the representation and tokenization of tasks, which would significantly impact the entire pipeline. Additionally, exploring different pre-trained language models, perhaps ones that include a Visual Transformer, could yield better results.

Some of these improvements I plan to implement during my time at INI. I will also write a detailed README for the GitHub repository to facilitate the continuation of this work.

Bibliography

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

[2] N. Dziri, X. Lu, M. Sclar, X. L. Li, L. Jiang, B. Y. Lin, P. West, C. Bhagavatula, R. L. Bras, J. D. Hwang, S. Sanyal, S. Welleck, X. Ren, A. Ettinger, Z. Harchaoui, and Y. Choi, "Faith and fate: Limits of transformers on compositionality," 2023.

[3] N. Butt, B. Manczak, A. Wiggers, C. Rainone, D. Zhang, M. Defferrard, and T. Cohen, "Codeit: Self-improving language models with prioritized hindsight replay," 2024.

[4] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, "Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions," 2019.

[5] M. Hodel, "Addressing the abstraction and reasoning corpus via procedural example generation," 2024.

[6] R. Cotterell, A. Svete, C. Meister, T. Liu, and L. Du, "Formal aspects of language modeling," 2024. [Online]. Available: https://drive.google.com/file/d/1IYgjs0Vf8TPmVW6w4S125j3G5Asatn4f/view

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[8] M. Phuong and M. Hutter, "Formal algorithms for transformers," 2022.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021.

[10] Y. Bengio and Y. Lecun, "Convolutional networks for images, speech, and time-series," 11 1997.

[11] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2023.

[12] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *CoRR*, vol. abs/1803.02155, 2018. [Online]. Available: http://arxiv.org/abs/1803.02155

[13] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," 2018.

[14] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," 2016.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[16] Google Cloud. (2024) Evaluate model performance with automl translation. Accessed: 2024-06-18. [Online]. Available: https://cloud.google.com/translate/automl/docs/evaluate

[17] P. von Platen, "How to generate text: using different decoding methods for language generation with transformers," https://huggingface.co/blog/how-to-generate, 2020, accessed: 2024-06-19.

Appendix A

Appendix

A.1 Task 00d62c1b

A.2 Results from the Stormer for Classification

Example for grid Level, Predicted some functions but not the one for it was looked for.

```
---------- Example Task ----------
F1 Score: 0.0
Average Precision: 0.0476190485060215
Accuracy: 0.9312499761581421
This comes from the dataset /home/jkleutgens/data_test/gl_schema
Correct predictions:
[]
Wrong predictions:
['fork', 'mapply', 'paint', 'sfilter', 'merge', 'lbind', 'objects',
    'rbind', 'compose', 'cover']
Forgotten to predict:
['trim']

The code of DSL functions in the task:
def solve_2mzktosx(
```

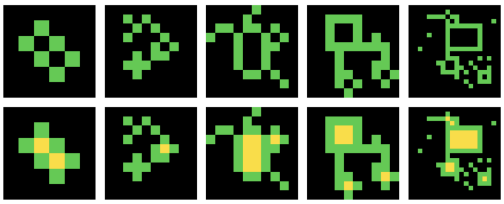


Figure A.1: task 00d62c1b

```
    I: Grid
) -> Grid:
    x1 = trim(I)
    O = trim(x1)
    return O
```

Example for Order Ranking

```
---------- Example Task ----------
F1 Score: 0.48000001907348633
Average Precision: 0.4588313400745392
Accuracy: 0.918749988079071
This comes from the dataset /home/jkleutgens/data_test/or_schema
Correct predictions:
['canvas', 'mostcolor', 'paint', 'objects', 'double', 'add']
Wrong predictions:
['fork', 'mapply', 'sfilter', 'merge', 'lbind', 'rbind', 'compose',
    'cover']
Forgotten to predict:
['astuple', 'shift', 'shape', 'argmin', 'normalize']

The code of DSL functions in the task:
def solve_a3mu97y0(
    I: Grid
) -> Grid:
    x1 = mostcolor(I)
    x2 = astuple(ZERO, ZERO)
    x3 = double(x2)
    x4 = objects(I, T, T, T)
    x5 = argmin(x4, width)
    x6 = shape(x5)
    x7 = add(x3, x6)
    x8 = canvas(x1, x7)
    x9 = normalize(x5)
    x10 = shift(x9, x2)
    O = paint(x8, x10)
    return O
```

A.3 Representation and Tokenization

Here is the tokenization shown of a solver:

```
['_de', 'f', '_solve', '_', '00', 'd', '62', 'c', '1', 'b', '(', 'I
   ', ')', ':', '_',  'x', '1', '_=', '_objects', '(', 'I', ',', '
   _T', ',', '_F', ',', '_F', ')',  '_', 'x', '2', '_=', '_color',
    'filter', '(', 'x', '1,', '_', 'ZE', 'RO', ')', '_', 'x', '3',
    '_=', '_', 'r', 'bind', '(', 'border', 'ing', ',', '_I', ')',
   '_', 'x', '4', '_=', '_compose', '(', 'f', 'lip', ',', '_', 'x
   ', '3)', '_', 'x', '5', '_=', '_', 'm',  'filter', '(', 'x',
   '2,',  '_', 'x', '4)', '_O', '_=', '_fill', '(', 'I', ',', '_F
   ', 'OUR', ',', '_', 'x', '5)', '_return', '_O', '</s>']
```

Here is an example of a tokenized grid:

```
' _Blackx28; _Blackx28; _Blackx28; _Blackx6 _Gray _Blackx21;
   _Blackx5 _Gray _Black  _Gray _Blackx14 _Gray _Black _Gray _Gray
    _Black _Black; _Blackx7 _Gray _Blackx15 _Gray _Black _Gray
   _Black _Black; _Blackx28; _Blackx28; _Blackx28; _Blackx28;
   _Blackx28; _Blackx28; _Blackx5 _Gray _Gray _Gray _Blackx20;
   _Blackx6 _Gray  _Gray _Blackx20; _Blackx28; _Blackx28;
   _Blackx28; _Blackx28; _Blackx28;  _Blackx28; _Blackx7 _Gray
   _Gray _Blackx19; _Blackx9 _Gray _Blackx18; _Blackx28; _Blackx28
   ; _Blackx28; _Blackx28; _Blackx28;'
```

A.4 Results from T5

A.4.1 Results 1.1

Example generated solver for a task in the Grid Level test set from the Base T5 model with "Variable to Alphabet" in the third epoch.

```
 Dataset /gl_schema
------------------- Random  Example  --------------------
Value of Blue Score: 0.0860814937735461
The Levenshtein distance is: 4844
Accuracy: 0.0
Code Generated an output: 1
The Number of Seen Pairs is: tensor(1)

Generated Text:
from dsl import *
from constants import *


def solve_1junml58(I: Grid) -> Grid:
```

```
    # there was an error in the code in this line
    # generated <extra_id_0>
    # generated ZERO
    x6 = rbind(astuple, x5)
    x7 = chain(x6, x6, partition)
    x8 = fork(paint, identity, x7)
    x9 = lbind(recolor, ZERO)
    x10 = compose(ulcorner, asobject)
    x11 = compose(urcorner, asobject)
    x12 = fork(connect, x10, x11)
    x13 = compose(x9, x12)
    x14 = fork(paint, identity, x13)
    x15 = chain(rot90, x14, rot270)
    x16 = x15(I)
    O = x8(x16)
    return O


Actual Text:
from dsl import *
from constants import *


def solve_1junml58(I: Grid) -> Grid:
    x1 = lbind(canvas, ZERO)
    x2 = chain(increment, increment, shape)
    x3 = compose(x1, x2)
    x4 = rbind(shift, UNITY)
    x5 = compose(x4, asobject)
    x6 = fork(paint, x3, x5)
    x7 = bottomhalf(I)
    O = x6(x7)
    return O


 predicted tokens: ['<pad>', '<extra_id_0>', 'ZERO', '#newline', '
    x6', 'rbind', ';', 'astuple', 'x5', '#newline', 'x7', 'chain',
    ';', 'x6', 'x6', 'partition', '#newline', 'x8', 'fork', ';', '
    paint', 'identity', 'x7', '#newline', 'x9', 'lbind', ';', '
    recolor', 'ZERO', '#newline', 'x10', 'compose', ';', 'ulcorner
    ', 'asobject', '#newline', 'x11', 'compose', ';', 'urcorner', '
    asobject', '#newline', 'x12', 'fork', ';', 'connect', 'x10', '
    x11', '#newline', 'x13', 'compose', ';', 'x9', 'x12', '#newline
```

```
    ', 'x14', 'fork', ';', 'paint', 'identity', 'x13', '#newline',
    'x15', 'chain', ';', 'rot90', 'x14', 'rot270', '#newline', 'x16
    ', 'x15', ';', 'I', '#newline', 'O', 'x8', ';', 'x16', '#EoF',
    '<pad>', ...


Actual tokens: ['#newline', 'x1', 'lbind', ';', 'canvas', 'ZERO',
    '#newline', 'x2', 'chain', ';', 'increment', 'increment', '
    shape', '#newline', 'x3', 'compose', ';', 'x1', 'x2', '#newline
    ', 'x4', 'rbind', ';', 'shift', 'UNITY', '#newline', 'x5', '
    compose', ';', 'x4', 'asobject', '#newline', 'x6', 'fork', ';',
     'paint', 'x3', 'x5', '#newline', 'x7', 'bottomhalf', ';', 'I',
     '#newline', 'O', 'x6', ';', 'x7', '#EoF', '<pad>', ...
```

A.4.2 Results 2.1

These results are drawn from the small T5 with "Variable to Alphabet" in the 6th epoch. And show that a different DSL solver

```
-------------------- Dataset /ct_schema 6--------------------
-------------------- Random --------------------
Value of Blue Score: 0.2484634754748475
The Levenshtein distance is: 4722
Accuracy: 1.0
Code Generated an output: 1
The Number of Seen Pairs is: tensor(0.7500, dtype=torch.float64)

Generated Text:
from dsl import *
from constants import *


def solve_2y0z78as(I: Grid) -> Grid:

    x1 = objects(I, T, T, T)
    x2 = rbind(equality, ZERO)
    x3 = lbind(compose, x2)
    x4 = lbind(chain, size)
    x5 = compose(dneighbors, last)
    x6 = rbind(x4, x5)
    x7 = lbind(lbind, intersection)
    x8 = compose(x7, toindices)
    x9 = compose(x6, x8)
```

```
    x10 = compose(x3, x9)
    x11 = fork(sfilter, identity, x10)
    x12 = compose(size, x11)
    x13 = fork(equality, size, x12)
    x14 = sfilter(x1, x13)
    x15 = merge(x14)
    x16 = cover(I, x15)
    x17 = difference(x1, x14)
    x18 = merge(x17)
    x19 = cover(x16, x18)
    x20 = mapply(hmirror, x14)
    x21 = recolor(TWO, x20)
    O = paint(x19, x21)
    return O


Actual Text:
from dsl import *
from constants import *


def solve_2y0z78as(I: Grid) -> Grid:

    x1 = objects(I, T, T, T)
    x2 = rbind(greater, ONE)
    x3 = compose(x2, height)
    x4 = compose(x2, width)
    x5 = fork(both, x3, x4)
    x6 = fork(connect, urcorner, llcorner)
    x7 = fork(equality, toindices, x6)
    x8 = fork(both, x5, x7)
    x9 = sfilter(x1, x8)
    x10 = merge(x9)
    x11 = cover(I, x10)
    x12 = difference(x1, x9)
    x13 = merge(x12)
    x14 = cover(x11, x13)
    x15 = mapply(vmirror, x9)
    x16 = recolor(TWO, x15)
    O = paint(x14, x16)
    return O


 predicted tokens: ['#BoF', '#newline', 'x1', 'objects', ';', 'I',
```

```
    'T', 'T', 'T', '#newline', 'x2', 'rbind', ';', 'equality', '
    ZERO', '#newline', 'x3', 'lbind', ';', 'compose', 'x2', '#
    newline', 'x4', 'lbind', ';', 'chain', 'size', '#newline', 'x5
    ', 'compose', ';', 'dneighbors', 'last', '#newline', 'x6', '
    rbind', ';', 'x4', 'x5', '#newline', 'x7', 'lbind', ';', 'lbind
    ', 'intersection', '#newline', 'x8', 'compose', ';', 'x7', '
    toindices', '#newline', 'x9', 'compose', ';', 'x6', 'x8', '#
    newline', 'x10', 'compose', ';', 'x3', 'x9', '#newline', 'x11',
     'fork', ';', 'sfilter', 'identity', 'x10', '#newline', 'x12',
    'compose', ';', 'size', 'x11', '#newline', 'x13', 'fork', ';',
    'equality', 'size', 'x12', '#newline', 'x14', 'sfilter', ';', '
    x1', 'x13', '#newline', 'x15', 'merge', ';', 'x14', '#newline',
     'x16', 'cover', ';', 'I', 'x15', '#newline', 'x17', '
    difference', ';', 'x1', 'x14', '#newline', 'x18', 'merge', ';',
     'x17', '#newline', 'x19', 'cover', ';', 'x16', 'x18', '#
    newline', 'x20', 'mapply', ';', 'hmirror', 'x14', '#newline', '
    x21', 'recolor', ';', 'TWO', 'x20', '#newline', 'O', 'paint',
    ';', 'x19', 'x21', '#EoF', '<pad>', ...


Actual tokens: ['#BoF', '#newline', 'x1', 'objects', ';', 'I', 'T',
     'T', 'T', '#newline', 'x2', 'rbind', ';', 'greater', 'ONE', '#
    newline', 'x3', 'compose', ';', 'x2', 'height', '#newline', 'x4
    ', 'compose', ';', 'x2', 'width', '#newline', 'x5', 'fork',
    ';', 'both', 'x3', 'x4', '#newline', 'x6', 'fork', ';', '
    connect', 'urcorner', 'llcorner', '#newline', 'x7', 'fork',
    ';', 'equality', 'toindices', 'x6', '#newline', 'x8', 'fork',
    ';', 'both', 'x5', 'x7', '#newline', 'x9', 'sfilter', ';', 'x1
    ', 'x8', '#newline', 'x10', 'merge', ';', 'x9', '#newline', '
    x11', 'cover', ';', 'I', 'x10', '#newline', 'x12', 'difference
    ', ';', 'x1', 'x9', '#newline', 'x13', 'merge', ';', 'x12', '#
    newline', 'x14', 'cover', ';', 'x11', 'x13', '#newline', 'x15',
     'mapply', ';', 'vmirror', 'x9', '#newline', 'x16', 'recolor',
    ';', 'TWO', 'x15', '#newline', 'O', 'paint', ';', 'x14', 'x16',
     '#EoF', '<pad>', ...
```