A Context-Aware Neural Network for the Abstraction and Reasoning Corpus (ARC)

Ahmed Abdelhamed
African Institute for Mathematical Sciences (AIMS)
ahmeda@aims.ac.za

Supervised by: Prof. Ulrich Paquet
AIMS, South Africa
Co-supervisor: Jaron Cohen
AIMS, South Africa

June 12, 2025
Submitted in partial fulfillment of AI for Science masters degree at AIMS South Africa

Abstract

Modern artificial intelligence models excel at pattern recognition but mostly fail at the kind of abstract reasoning that humans use to solve novel problems from few examples. The Abstraction and Reasoning Corpus (ARC) was created to measure this gap in AI capabilities. To address this challenge, this thesis introduces a two-stage neural architecture designed to mimic a human approach to problem-solving. First, a Vision Transformer encoder analyses a few demonstration grids to distil a puzzle's underlying abstract rule into a compact "context" representation. Second, a conditional U-Net decoder uses this context to guide the pixel-perfect transformation of a new, unseen input grid. By training a specialised model for each unique puzzle, the system is forced to learn the specific logic from the handful of examples provided. This approach proves effective, perfectly solving 78 of the 833 ARC-AGI-2 public training tasks and achieving a mean pixel-level accuracy of 73.8% and a Kaggle Pass@2 score of 0.83% on the private test set. This work provides a strong case that learning abstract rules from context is a viable path for solving tasks that require reasoning.

Declaration

I, the undersigned, hereby declare that the work contained in this research project is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Ahmed Eldaw Abdelhamed, 13 June 2025

Contents

1. Introduction

1.1 The Search for Thinking Machines

For decades, the field of Artificial Intelligence has been driven by a single, bold ambition: to build a machine that can think. This quest was given its modern framing not as a fantasy, but as a concrete engineering problem by Alan Turing, who set the challenge with his inquiry into "Computing Machinery and Intelligence" Turing (1950). The field's founding core belief was soon codified at the 1955 Dartmouth workshop, where its pioneers made a now-famous conjecture: that every facet of intelligence could be so precisely described that a machine could be built to simulate it McCarthy et al. (2006). This was the birth of the search for Artificial General Intelligence (AGI) an intellect defined not by its narrow skill, but by its fluid, adaptive, and general purpose competence.

Fast forward to today, and one could be forgiven for thinking this goal is within reach. The landscape of AI has been completely transformed by the eruption of deep learning, a paradigm that has delivered staggering results Goodfellow et al. (2016). Deep neural networks have achieved decisive, superhuman mastery in domains of immense strategic complexity like Go Silver et al. (2016); they have shattered the benchmarks for visual recognition on datasets like ImageNet Alex Krizhevsky (2012); and with the arrival of architectures like the Transformer, they have revolutionised our expectations for machine understanding of human language Vaswani et al. (2017). This is not a story of incremental progress. It is the story of a brute-force convergence of algorithmic insight, massive data, and exponential computation that has redefined the state-of-the-art LeCun et al. (2015).

However, the nature of this success also highlights a significant limitation in the current deep learning paradigm. The power of these models stems from their effectiveness at finding and exploiting statistical correlations in data, yet this is also a primary source of their weakness. These systems are well-known for their brittleness; their performance can degrade significantly when faced with problems even slightly outside their training distribution Brenden M. Lake (2016). For all their complexity, their decision-making processes can be opaque and their grasp of underlying concepts is often considered shallow Marcus (2018). This reliance on statistical correlation, rather than a robust conceptual understanding, suggests that further progress towards AGI is unlikely to be achieved by simply scaling existing models. It demands a renewed focus on first principles and on instilling the core faculties of abstraction and reasoning which current architectures often lack.

1.2 The Core Deficit: Abstraction and Reasoning

To bridge the chasm towards AGI, we must dissect the specific cognitive faculties that are absent in our most powerful models. The critical missing pieces are **abstraction** and **reasoning** the twin capabilities that support the efficiency and adaptability of human thought. These are not simply advanced forms of pattern matching. They are the deeper cognitive engines for creating and manipulating knowledge.

Reasoning, in any meaningful sense, must transcend statistical correlation. It is the process of forging conclusions and plans from facts and premises, allowing a model to operate beyond its explicit training and derive new knowledge to achieve its goals Russell and Norvig (2020). Abstraction, conversely, is the cognitive mechanism by which general rules and concepts are derived from specific examples. It involves identifying essential patterns, distilling a conceptual model from sparse and noisy data, and representing that knowledge in a way that supports generalisation Tenenbaum et al. (2011). It is the mechanism by which humans effortlessly form concepts like "tool" or "obstacle", which are defined not by their

pixel-level appearance but by their role in a given context, allowing for the powerful generalisation of knowledge to entirely new situations Brenden M. Lake (2016). These faculties are deeply connected, and the entire history of AI can be read as a struggle to unite them. The early symbolic paradigm, in its pursuit of "Good Old-Fashioned AI" or GOFAI Haugeland (1985) built systems with formidable deductive reasoning engines but required that all necessary abstractions (the symbols and rules) be carefully hand-crafted by human experts McCarthy et al. (2006). This approach proved fatally brittle, as these systems could not form their own concepts when faced with the ambiguity of the real world Brachman (1985). The connectionist revolution offered the inverse: systems that excel at learning representations, a form of abstraction directly from sensory data, yet lack any capacity for robust, systematic reasoning over those learned concepts.

The failure to unify learned abstraction with versatile reasoning is a widely acknowledged demarcation between modern AI and true, general intelligence. The central argument of this thesis is that a two-stage, context-aware neural architecture can effectively solve a range of abstract reasoning tasks by learning to unify these faculties within a single, specialised model. Abstraction is the prerequisite for all efficient reasoning; without it, a model is trapped, forced to treat every new problem as if it were the first. The few-shot learning demands of a benchmark like the Abstraction and Reasoning Corpus (ARC) are so powerful because they are designed to break any system that has not solved this intertwined challenge. Success is impossible without both abstracting a rule from a handful of examples and then reasoning with that rule to solve a new instance Chollet (2019). The chasm between machine and human performance on ARC is a measure of this core deficit. This is a deficit that this thesis directly confronts.

1.3 A Historical Perspective on AI's Pursuit of Reasoning

The history of artificial intelligence can be broadly understood as a tale of two competing philosophies for building a reasoning machine: the symbolic and the connectionist paradigms.

1.3.1 The Symbolic Era

The first real push to create reasoning machines was through "Good Old-Fashioned AI". The core belief was simple but powerful: intelligent thought is nothing more than manipulating symbols according to a set of formal rules, a concept known as the physical symbol system hypothesis Newell and Simon (1976). This idea sent researchers down two main paths. One path, led by John McCarthy, aimed to give programs "common sense" by using the language of formal logic. His "Advice Taker" concept was for a machine that could be told facts about the world and then logically deduce what to do next, kicking off the field's long-standing focus on explicit knowledge McCarthy et al. (2006).

At the same time, Allen Newell and Herbert Simon were working on general problem-solving methods. Their famous program, the General Problem Solver (GPS), was a landmark because it tried to separate the strategy of solving a problem from the specific knowledge about that problem, a key step towards a more universal approach to thinking Ernst and Newell (1969).

This symbolic approach had its heyday with the "expert systems" of the 1970s and 1980s. Programs like DENDRAL, which could identify chemical structures, and MYCIN, which diagnosed blood infections, showed that this method could work wonders in very specific areas. They did this by encoding the knowledge of human experts into huge sets of "if-then" rules Feigenbaum (1977). The whole process of extracting this knowledge from experts and painstakingly programming it became a field called

"knowledge engineering." Languages like Prolog were built for exactly this, allowing developers to write programs as a collection of logical facts and rules Colmerauer and Roussel (1996).

The strength of this symbolic approach was its clarity. You could see the knowledge, and you could trace the logical steps the machine was taking. But this was also its fatal flaw. These systems were incredibly "brittle"; if they encountered a situation that was not covered by their pre-programmed rules, they would simply break Dreyfus (1972). They were also stuck in a "knowledge acquisition bottleneck." Manually programming all the common sense knowledge that humans take for granted turned out to be a nearly impossible task Feigenbaum (1977). The dream of building a general intelligence by hand-writing rules collided with the messy reality of the world, leading to a loss of faith and funding known as the "AI Winters" Lighthill (1973). The symbolic era proved that reasoning could be mechanised, but its inability to learn or adapt on its own meant that a new approach was desperately needed.

1.3.2 The Connectionist Revolution: A New Way of Thinking

The old way of doing AI hit a dead end. Symbolic systems could not handle the messiness of the real world, so researchers started looking for a new approach: connectionism. The idea was to stop programming rules by hand and instead build systems that learned from data, inspired by the network of neurons in the human brain Feldman and Ballard (1982). Early work by McCulloch and Pitts showed that simple neuron-like units could perform logic McCulloch and Pitts (1943), and Frank Rosenblatt's Perceptron was an exciting first try at a trainable network Rosenblatt (1958). But the excitement did not last. A famous critique by Minsky and Papert proved that these simple networks had severe limitations, and for years, nobody had a good way to train bigger, more powerful ones Minsky and Papert (1969). The field was stuck.

The big breakthrough that changed everything was an algorithm called backpropagation. Though it was invented years earlier by Paul Werbos Werbos (1974), it was a 1986 paper by Rumelhart, Hinton, and Williams that really showed the world how to use it Rumelhart et al. (1986). Suddenly, researchers had a key. Backpropagation gave them a way to efficiently train deep networks by feeding error signals backward to tweak all the connections. This unlocked the potential for AI to finally learn complex patterns on its own.

With this new tool, the modern fundamental block of AI were born. For images, Convolutional Neural Networks (CNNs) became the standard, using a clever design pioneered by Yann LeCun to process visual information (LeCun et al., 1989). For anything in a sequence, like speech or text, Long Short-Term Memory (LSTM) networks were the answer for a long time Hochreiter and Schmidhuber (1997). And more recently, the Transformer architecture has taken over completely, using a powerful trick called "self-attention" to understand language at a massive scale (Vaswani et al., 2017). This explosion of new ideas, combined with huge datasets like ImageNet Deng et al. (2009) and the raw power of modern computer chips (GPUs), created the deep learning revolution we see today a moment perfectly captured by the victory of a system called AlexNet in 2012 Alex Krizhevsky (2012).

But this new way came with a trade-off. Its greatest strength is that it can learn subtle patterns from huge amounts of raw data, which is exactly where the old symbolic AI failed. The problem, however, is that this is essentially all it does. These networks are often brittle, and can fail in strange ways when they see something slightly new Marcus (2018). They do not "think" in a way we can understand, and they can not easily combine old ideas to make new ones. They are also incredibly data-hungry, needing to see millions of examples to learn something a human can from just one or two. Worst of all, they are "black boxes". It is nearly impossible to know for sure why a network makes a certain decision, which is a significant problem for building AI we can actually trust Pearl and Mackenzie (2018). The

connectionist revolution solved the problems of the old symbolic systems, but it gave us a whole new set of equally hard problems to solve.

1.3.3 Hybrid Systems: A New Hope

This history leads to a clear and unavoidable conclusion: neither of the two great AI paradigms, on its own, is going to get us to AGI. Symbolic AI was too rigid, and connectionist AI is too shallow. This realisation has pushed the research community towards a new, common sense idea: if each approach has strengths and weaknesses, why not try to build hybrid systems that combine the best of both worlds?

The most promising of these new ideas is called **neuro-symbolic AI**. The goal is simple: create a single system that gets the powerful, data-driven learning of neural networks and the clean, logical reasoning of symbolic systems d'Avila Garcez and Lamb (2023). The idea is to let the neural network do what it is good at, learning from messy, unstructured data like images while letting a symbolic component handle what it is good at, like step-by-step reasoning and using prior knowledge Hassabis et al. (2017). This is part of a broader trend called "informed machine learning", where researchers are finding clever ways to inject existing human knowledge into learning systems instead of forcing them to learn everything from scratch von Rueden et al. (2023).

Researchers are moving away from treating one paradigm and are instead using the right tool for the right job. But this is much easier said than done. The biggest challenge is figuring out how to make these two fundamentally different ways of thinking talk to each other. Neural networks think in terms of continuous numbers (vectors), while symbolic systems think in discrete facts and rules. Getting them to connect in a meaningful way is a massive technical and conceptual hurdle Marcus (2020). The complexity of this challenge is precisely why we need new benchmarks and new architectures to test these new, integrated ideas and see if they can finally overcome the limitations that have held back AI for decades.

1.4 The Abstraction and Reasoning Corpus

If we want to build an AI that can truly reason and abstract, we need a way to measure it. We need a benchmark that does not just test for skill on a single task, but that probes for the very cognitive abilities we care about. The Abstraction and Reasoning Corpus (ARC), introduced by Franccois Chollet, was created for exactly this purpose Chollet (2019). ARC is not just another dataset. It was built with a specific philosophy in mind: to measure what psychologists call "fluid intelligence" the ability to solve new problems on the fly, without relying on prior knowledge or extensive training Cattell (1963).

This idea cuts directly against the grain of most modern AI benchmarks. Many famous benchmarks can be beaten by simply training a bigger model on more data, which allows a system to "buy" skill through brute force rather than demonstrating true generalisation Chollet (2019), ARC makes this much harder. It consists of few unique visual reasoning puzzles, but for each puzzle, a system is only shown a tiny number of examples sometimes just a single input-output pair. This extreme "few-shot" setting means that rote memorisation and simple pattern matching are useless. To succeed, a system has no choice but to figure out the underlying abstract rule and apply it to a new situation.

Furthermore, ARC is designed to test for the kind of "core knowledge" that humans seem to be born with. These are the basic, intuitive concepts about the world like the idea that objects are solid and do not just disappear, or a basic sense of numbers and shapes that children grasp with very little experience Spelke and Kinzler (2007). Current AI systems do not have this built-in common sense, and many of

ARC's seemingly simple puzzles are impossible to solve without it. This explains the huge gap between how people and machines perform on ARC. Humans find most tasks straightforward, while even the most powerful AI models today struggle mightily.

1.5 Approaches to Solving ARC

The difficulty of the ARC benchmark is best understood through the major research initiatives organised to solve it. The ARC Prize is the foremost competition, offering a $700,000 Grand Prize for a system that can achieve 85% accuracy on the private test set ARC Prize, Inc. (2025). The competition is hosted on Kaggle with strict efficiency limits, requiring solutions to run offline and without access to external APIs like GPT-4 or Claude, ensuring that progress comes from genuine, self-contained reasoning ability.

The approaches that have shown the most success can be grouped into two main categories: symbolic methods and uses of Large Language Models (LLMs).

1.5.1 Symbolic Search

The foundational and most successful strategy is **program synthesis** using a custom-built Domain Specific Language (DSL). This method involves designing a small, specialised programming language with functions tailored for ARC, such as **move_object** or **recolour_shape**. The system then performs a massive search to find a sequence of these functions that solves the puzzle.

A well-documented example of this pure symbolic DSL strategy is the winning solution from the first ARC competition in 2020 by *icecuber*. In his technical write-up, he details the creation of his DSL that his system searches over icecuber (2020).

1.5.2 LLM-Driven Code Generation and Test-Time Training

On the other hand, the most significant breakthroughs of 2024 occurred on the public leaderboard, where teams used Large Language Models (LLMs) to generate code in a general-purpose language like Python.

The highest scoring of these is Jeremy Berman's **Evolutionary Test-time Compute**, which achieved 53.6%. This method uses an LLM to generate hundreds of complete Python functions. The best performing functions are then "evolved" by feeding them back to the LLM to create better variations, a process that continues until a working solution is found Berman (2024).

A second successful LLM-based method is **Test-Time Training (TTT)**, which scored 47.5% and was developed by a joint team from MIT and Cornell ARC Prize, Inc. (2024). Here, a language model is briefly fine-tuned on a specific task's examples *at the moment of testing*, allowing it to adapt its knowledge to the immediate problem Akyürek et al. (2024).

1.5.3 Positioning of this Thesis

It is critical to note that both the leading symbolic DSL approach and the innovative LLM-based methods from 2024 are ultimately focused on finding or generating an explicit, step-by-step programme. They represent a sophisticated search for an algorithm, with the LLMs acting as powerful guides or generators for that search.

The work presented in this thesis explores a fundamentally different path. We do not use a DSL, generate code, or perform a search for an explicit programme. Our aim is to investigate whether a purely neural architecture, operating directly on pixels, can learn the necessary abstract transformations implicitly.

1.6 Our Proposed Solution

This thesis introduces a new architecture designed specifically for ARC challenges. The core idea is inspired by In-Context Learning (ICL), a paradigm where a model learns to perform a novel task by conditioning on a few examples provided on the fly Tom B. Brown (2020). Our approach adapts this philosophy to a specialised system: for each ARC puzzle, we train a new model from scratch where the few demonstration pairs constitute the entire training set.

The model we develop in this thesis is a two-step system designed to mimic how a person might solve one of these puzzles. First, it tries to understand the rule. Second, it applies that rule to solve the problem.

The first step (the abstraction) is handled by a Vision Transformer ViT, which acts as an encoder Dosovitskiy et al. (2021). The ViT's job is to look at the few "input-output" demonstration grids that define a puzzle. ViT processes these pairs and distils the underlying transformation logic into a single, compact vector of numbers. We call this the "transformation embedding" or "context vector." This vector is a learned summary of the puzzle's hidden rule. The use of a Vision Transformer is motivated by other neural investigations into ARC, such as the work by Li et al. (2025), which highlighted the effectiveness of ViTs for these tasks.

The second step (the reasoning) is handled by a conditional U-Net, which acts as a decoder Ronneberger et al. (2015). The U-Net takes the new test grid that needs to be solved. But crucially, its decision making process is guided by the context vector it receives from the ViT. This contextual guidance, which is injected into the U-Net at multiple points using cross-attention, allows the decoder to apply the specific rule it was just given.

The choice of these two components was deliberate. Vision Transformers are well-suited for capturing the global relationships required to infer a puzzle's underlying rule, while U-Nets are specialised for image-to-image tasks that demand precise, pixel-level generation. This section is just a high level glimpse; the detailed breakdown of how each component works is reserved for later chapters.

1.7 Contributions and Thesis Roadmap

This thesis makes the following primary contributions to the study of abstract reasoning on the ARC benchmark:

1. **A New Architecture:** We are proposing, building, and testing a new system. It is a two part model made of a Vision Transformer and a U-Net, with attention blocks, designed from the ground up to tackle the kind of "learn from examples" visual puzzles found in ARC.
2. **A Puzzle-by-Puzzle Training Method:** Instead of training one giant model, we train a fresh, specialised model for every single puzzle. This forces the system to learn the specific logic for that one task, which is a core part of our approach.
3. **Qualitative Analysis:** We do not just show the final scores. We dig deep into the results to see exactly which kinds of puzzles the model we develop can solve and, more importantly, which ones

it fails on. This gives us a much clearer picture of what the model is actually capable of thinking.

The rest of this thesis is organised to walk the reader through this research step by step. We will begin by providing a detailed look at the ARC benchmark itself, explaining its structure and why it is such a powerful tool for measuring intelligence. Following that, we will break down the model architecture, detailing the design of both the Vision Transformer encoder and the conditional U-Net decoder. Next, we will outline the methodology, including the data preparation and the task-specific training pipeline. then, we will present and analyse the results of the experiments, showing both the quantitative performance and a qualitative look at specific puzzles.

2. Abstract and Reasoning Problem

2.1 What Is ARC?

The Abstraction and Reasoning Corpus (ARC), proposed by Chollet (2019), is a benchmark designed to test artificial general intelligence (AGI) beyond simple pattern recognition. Unlike traditional computer vision tasks, which often rely on large datasets and local feature extraction, ARC tasks require a model or a system that can infer abstract transformation rules from very few examples and generalise them perfectly to novel inputs we call this model solver. Relying on local texture cues is insufficient because ARC transformations often depend on global properties; a successful solver must grasp the global concept linking an input grid $\mathbf{X}$ to its output grid $\mathbf{Y}$, as even a single pixel error invalidates a solution.

Humans typically solve ARC puzzles because we already have "core knowledge priors" an intuitive grasp of objects, symmetry, basic counting, and simple cause-effect relations. These priors help us spot the high-level rules that map $\mathbf{X}$ to $\mathbf{Y}$, in a recent ARC-AGI-2 experiment, 400 volunteers solved 1,417 unique tasks, every task in about 2.3 minutes on average and needed no more than two attempts per puzzle. In contrast, Claude Opus 4, the SOTA from anthropic even with "Thinking 16k" achieves only 8.6% accuracy on the same benchmark. This wide gap highlights how much progress is still needed before models can match the abstract reasoning that comes naturally to humans.

2.2 Data

The core challenge of generalisation in ARC, introduced in Section 2.1, is fundamentally shaped by how the data is presented for each task. To understand this, we can look at the example puzzle with ID `0ca9ddb6`, shown in Figure 2.1. In the first training pair $\mathbf{X}_1, \mathbf{Y}_1$, the input grid $\mathbf{X}_1$ contains two blue squares, two red squares, and one cyan square. In the output grid $\mathbf{Y}_1$, a clear transformation pattern can be seen: the red squares are covered by 4 diagonal yellow squares (placed directly over the same position), the blue squares are covered by 4 orange cross squares, and the cyan square stays exactly the same. This same rule is applied in the second training pair $\mathbf{X}_2, \mathbf{Y}_2$, where only red and blue squares appear in $\mathbf{X}_2$, and they are transformed in $\mathbf{Y}_2$ using the same yellow diagonal and orange cross patterns. The transformation depends entirely on the colour of each square and applies the same logic consistently.

In general, each ARC task provides a small set of such training examples $\mathbf{X}_k, \mathbf{Y}_k$ that follow a consistent, but hidden, transformation rule. The solver's job is to discover this rule and apply it to the unseen test input $\mathbf{X}_{\text{test}}$ to produce the correct output $\mathbf{Y}_{\text{test}}$. This focus on achieving strong generalisation from minimal data is what makes ARC a challenging and interesting benchmark for abstract reasoning.

So, each ARC task is a self-contained puzzle. It provides a small set of these training examples, where a single training pair consists of one input grid $\mathbf{X}$ and its corresponding ground-truth output grid $\mathbf{Y}$ for that specific puzzle. The grids are two-dimensional arrays of integers, where each integer from $C = \{0, \ldots, 9\}$ represents one of ten distinct colours. Formally, $\mathbf{X}, \mathbf{Y} \in \{0, \ldots, 9\}^{H \times W}$, where H and W denote height and width respectively.

While individual ARC tasks are self-contained, the broader ARC benchmark (ARC-AGI-2 dataset Section 2.1) comprises a large collection of such puzzles, it contains 1 000 public tasks for training, 120 for evaluation, and 240 for test.

The number of training pairs, for a given single task K, is deliberately small ($K \in \{1, \ldots, 4\}$, often

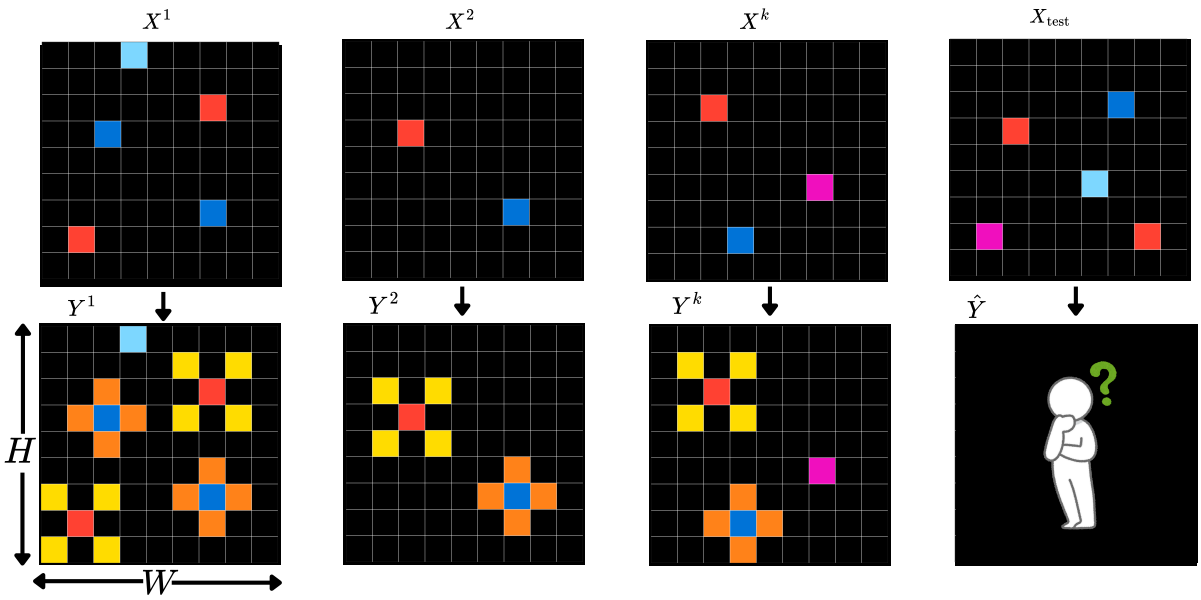


Figure 2.1: This figure illustrates the structure of an ARC-AGI task, for challenge with ID `0ca9ddb6`. The example shows K training input $\mathbf{X}^{(k)}$ / output $\mathbf{Y}^{(k)}$ pairs, followed by a test input $\mathbf{X}_{\text{test}}$ for which the solver must predict the corresponding output $\hat{\mathbf{Y}}$. Each grid is of size 9×9, where H represents the height (9 rows) and W represents the width (9 columns). The goal is to infer the underlying transformation rule from the training examples and apply it to the test input.

$K = 3$), emphasizing the few-shot learning aspect. For evaluation, each task includes $M \in \{1, 2\}$ test inputs $\mathbf{X}_{\text{test}}$ for which the model must predict the output $\mathbf{Y}_{\text{test}}$. Grid dimensions vary but never exceed 30×30 pixels.

2.3 Evaluation

The evaluation of performance on the Abstraction and Reasoning Corpus (ARC), particularly within the context of the ARC-AGI-2 benchmark on Kaggle, relies on a stringent scoring mechanism that necessitates exact generalisation. The dataset is organised into distinct partitions, comprising a public evaluation set and a private evaluation set containing 240 tasks. The private tasks are managed by Kaggle to prevent direct model training on their content.

The primary metric for the ARC-AGI-2 is Pass@2 accuracy. This metric is designed to accommodate the iterative nature in human problem-solving for ARC tasks. For each test input within a given task, a submission is afforded two distinct prediction attempts to precisely match the ground-truth output. A prediction for a single test input is awarded a score of 1 if either of its two attempts achieves an exact match with the ground-truth output grid, requiring identical pixel values and precise dimensional congruence. Otherwise, the score for that test output is 0.

For the final score they take the best (highest) score for each test grid across every task, add them all up, and then divide by the total number of test grids. This process can be formally expressed as:

$$\text{Accuracy} = \frac{1}{N_{\text{total}}} \sum_{i=1}^{N_{\text{tasks}}} \sum_{j=1}^{M_i} \max\Big(\mathbb{I}[\text{attempt}_{i,j,1} = \text{groundtruth}_{i,j}],\ \mathbb{I}[\text{attempt}_{i,j,2} = \text{groundtruth}_{i,j}]\Big) \tag{2.3.1}$$

N_{total}	The cumulative count of test outputs across all tasks in the evaluation set,
N_{tasks}	The total number of tasks included in the evaluation set,
M_i	The number of test outputs corresponding to task i,
$\mathbb{I}(\cdot)$	is the indicator function, which yields 1 if the condition is true (exact match) and 0 otherwise,
$\text{attempt}_{i,j,k}$	refers to the k^{th} prediction attempt for the j^{th} test output of task i,
$\text{groundtruth}_{i,j}$	represents the actual ground truth output for the j^{th} test output of task i.

2.3.1 Exact-Match Evaluation

Beyond the broad benchmark assessment, the fundamental criterion for judging any solution's success on an individual ARC puzzle's test case is remarkably stringent: exact matching. For a given test input $\mathbf{X}_{\text{test}}$, a predicted output grid $\hat{\mathbf{Y}}$ is considered correct only if it is an absolute pixel-perfect and shape-perfect replica of the corresponding ground-truth output grid $\mathbf{Y}_{\text{test}}$. This means every pixel must have the identical colour value, and the dimensions (height and width) of the predicted grid must precisely match those of the ground truth. Even a single incorrect pixel or a subtle difference in shape renders the prediction incorrect.

To quantify the performance on a single ARC puzzle, given its M test inputs, we define the puzzle-level accuracy, $\text{Acc}_{\text{puzzle}}(\cdot)$. This function measures the proportion of test cases within that specific puzzle that achieve an exact match:

$$\text{Acc}_{\text{puzzle}}(\{\hat{\mathbf{Y}}^{(m)}\}_{m=1}^{M}, \{\mathbf{Y}^{(m)}\}_{m=1}^{M}) = \frac{1}{M} \sum_{m=1}^{M} \mathbb{I}[\hat{\mathbf{Y}}^{(m)} = \mathbf{Y}^{(m)}] \tag{2.3.2}$$

2.3.2 Key Notations

Symbol	Description
C	Set of ten distinct colour indices, $\{0, \ldots, 9\}$
H, W	Original height and width of a grid
K	Number of training pairs for a given task (typically $K \in \{1, \ldots, 4\}$)
M	Number of test inputs for a given task (typically $M \in \{1, 2\}$)
$(\mathbf{X}^{(k)}, \mathbf{Y}^{(k)})$	The k-th training input grid and its corresponding ground-truth output grid
$\mathbf{X}_{\text{test}}$	A test input grid for which a prediction is required
$(\mathbf{Y}_{\text{test}})$	The ground-truth output grid corresponding to a test input
$(\hat{\mathbf{Y}})$	The solver's predicted output grid for a test input
N_{total}	Cumulative count of all individual test outputs across all tasks in the evaluation set
N_{tasks}	Total number of distinct tasks included in the evaluation set
$\mathbb{I}(\cdot)$	Indicator function, evaluating to 1 if the condition is true (exact match), and 0 otherwise
$\text{attempt}_{i,j,k}$	The k-th prediction attempt for the j-th test output of task i
$\text{Acc}_{\text{puzzle}}(\cdot)$	Puzzle-level accuracy based on exact matches

Table 2.1: Key Notations

3. System Architecture: The ViT Encoder

3.1 Overall System Architecture

As established in the previous chapter, the Abstraction and Reasoning Corpus (ARC) requires a model that can infer abstract rules from only a handful of examples.

Our architecture is a direct implementation of the In-Context Learning (ICL) paradigm, first introduced in Chapter 1. For each individual ARC puzzle, we construct and train a new, specialised model entirely from scratch using only its handful of demonstration examples. This approach is distinct from more common few-shot learning strategies, which typically involve pre-training a single, large model on a diverse range of tasks and then fine-tuning it on a new puzzle's examples. By intentionally training a fresh model for every task, we force the system to derive the transformation logic exclusively from the immediate context provided. Each model, while sharing the same underlying architecture, therefore becomes a specialist at inferring and applying the unique rules for its specific puzzle.

The architecture, illustrated in Figure 3.1, is composed of two distinct stages that work in sequence: a Transformation Encoding stage and a Conditional Output Generation stage.

Stage 1: Transformation Encoding. The first stage is handled by a Vision Transformer (ViT) encoder. Its sole responsibility is to learn the underlying logic of the puzzle. It does this by processing each example pair consisting of an input grid $\mathbf{X}$ and its corresponding output grid $\mathbf{Y}$ from the puzzle's training set. The ViT distils the transformation rule it observes into a compact numerical representation called a "transformation embedding" $\mathbf{e}$. This embedding is the key piece of context that will be passed to the next stage.

Stage 2: Conditional Output Generation. The second stage features a U-Net-based decoder. This stage takes a new, unseen test grid $\mathbf{X}_{\text{test}}$ and uses the transformation embeddings generated by the encoder to guide the creation of the final output. By conditioning its generation process on this context, the decoder can apply the specific rules it was just given. The full details of the decoder architecture will be explored in Chapter 4.

3.2 Vision Transformer for ARC-AGI

The Abstraction and Reasoning Corpus (ARC) challenges models to infer abstract grid transformations from a very limited number of examples. As discussed in Chapter 2, ARC tasks demand a solver capable of understanding global structures and relationships within grids, such as the one illustrated in Figure 2.1, as relying solely on local texture cues is insufficient to generalise correctly.

To address the ARC-AGI benchmark, we develop a model capable of learning abstract transformations from just a few examples. Our methodology, detailed in Chapter 5, constructs a dedicated model for each individual ARC puzzle. This approach is designed to infer a puzzle-specific transformation rule, rather than a universal rule across the entire dataset.

The Transformer, introduced by Vaswani et al. (2017) Vaswani et al. (2017), is now everywhere in deep learning, especially when one needs to turn one sequence into another. What makes it special is its "attention" mechanism: instead of processing data step by step, it looks at the whole input in one shot and determines, on the fly, which bits matter most. Because it can consider every element

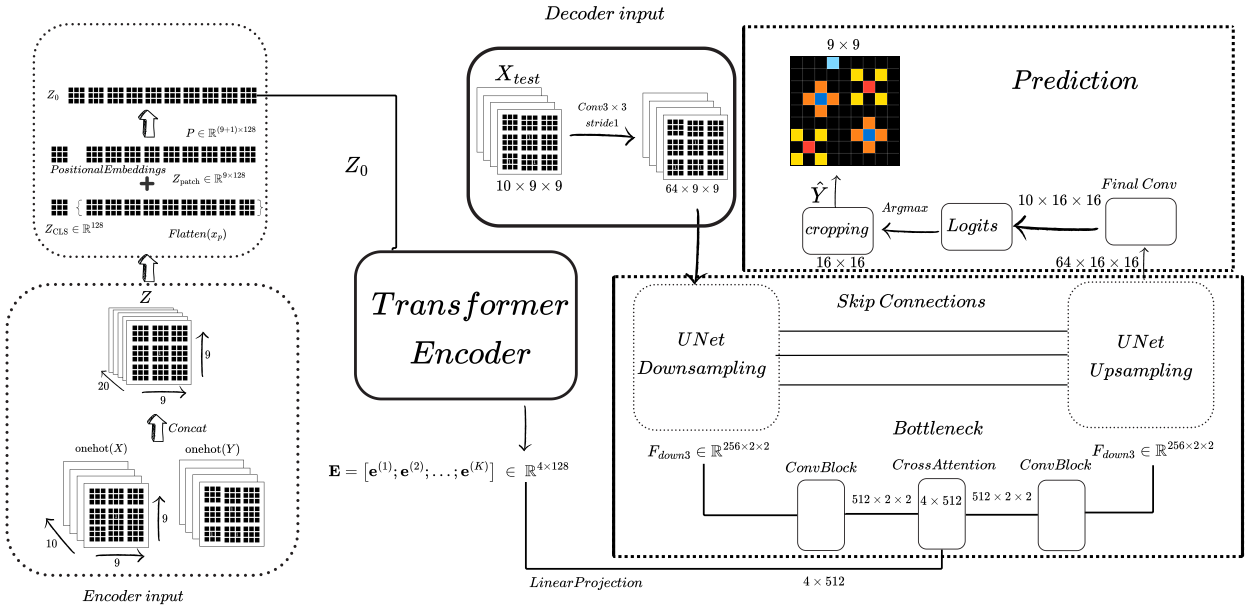


Figure 3.1: This figure illustrates the overall architecture of the In-Context Learning (ICL) system developed for ARC puzzle solving. The left panel details the Transformation Encoding stage: concatenated one-hot encoded training pairs $\mathbf{X}_k$,$\mathbf{Y}_k$, forming, for instance, a $20 \times 9 \times 9$ tensor, are processed by a Vision Transformer (ViT) encoder. This involves patchification, linear projection, and the addition of a CLS token and positional embeddings (yielding token embeddings in $\mathbb{R}^D$ where D=128 to produce a context matrix $\mathbf{E}$ ($\mathbb{R}^{K\times 128}$) of transformation embeddings. The right panel depicts the Conditional Output Generation stage: a test input grid $\mathbf{X}_{\text{test}}$ (9×9) is passed through an initial 3×3 convolution and then a U-Net decoder. The U-Net, conditioned on ($\mathbf{E}$) via cross-attention mechanisms and utilising skip connections, generates logits ($10 \times 16 \times 16$ grid). These are converted to the final predicted output grid $\hat{\mathbf{Y}}$ (9×9) through an argmax and cropping.

simultaneously, it is both fast and great at spotting long-range patterns and subtle connections that other models often miss.

The Vision Transformers (ViT) Dosovitskiy et al. (2021), extends this powerful Transformer architecture to the domain of image recognition. Instead of looking at every pixel one by one, a ViT breaks the image into small, non-overlapping "patches" that can think of as little tiles that cover the grid. What makes ViTs so powerful for ARC puzzles is their self-attention mechanism: Each patch can 'see' and weigh information from every other patch at the same time. In other words, the model does not just focus on local corners or edges; it can pick up patterns that span the entire grid. That is exactly the kind of global reasoning ARC demands.

In our approach, we use the ViT to process individual training pairs. Specifically, we feed the ViT an input grid $\mathbf{X}$ along with its corresponding ground-truth output $\mathbf{Y}$ from a given training example, as shown in Figure 2.1. The ViT is then trained to produce a single, compact "signature" (an embedding) that encapsulates the transformation rule mapping $\mathbf{X}$ to $\mathbf{Y}$.

3.3 Input Representation for the Vision Transformer (ViT)

ARC puzzles come in all sorts of grid shapes and sizes. Before we can feed them into the Vision Transformer (ViT) encoder, we need to convert these raw grids into a single, consistent format. In this section, we will explain exactly how we represent and preprocess ARC grids so they are ready for the model.

3.3.1 Grids and One-Hot Encoding

Recall from Chapter 2 that an ARC input grid $\mathbf{X}$ and its corresponding ground-truth output grid $\mathbf{X}$ are simply two-dimensional arrays of integers. Each cell at coordinates (i, j) (within dimensions $H \times W$) contains a number from

$$C = \{0, \ldots, 9\} \tag{3.3.1}$$

where each integer represents one of the ten colours of ARC.

For neural networks to correctly interpret these colour values, we first convert each integer into a 10-layer binary format using one-hot encoding. Why? If we kept raw numbers like 1, 2, or 9, the network could get the wrong idea thinking that colour 9 is "bigger" or "more important" than colour 1. One-hot encoding remedies this by giving each of the ten ARC colours its own independent channel, so no unintended ordering or hierarchy sneaks into the model.

Formally, for a grid $\mathbf{X}$, its one-hot encoded representation is defined as:

$$\mathrm{onehot}(\mathbf{X})[c, i, j] = \begin{cases} 1 & \text{if } X[i, j] = \mathbf{c}, \\ 0 & \text{otherwise,} \end{cases} \qquad \mathbf{c} \in \{0, \ldots, 9\} \tag{3.3.2}$$

After one-hot encoding, an input grid $\mathbf{X}$ of size $H \times W$ becomes a $(10 \times H \times W)$ binary tensor written as $\mathrm{onehot}(\mathbf{X}) \in \{0, 1\}^{10 \times H \times W}$. We do the same for the target grid $\mathbf{Y}$, getting $(\mathrm{onehot}(\mathbf{Y}) \in \{0, 1\}^{10 \times H \times W})$.

3.3.2 Padding and Consistent Sizing

ARC grids show significant variability in their dimensions, both within and across puzzles. To ensure compatibility with batch processing and the architectural requirements of the Vision Transformer (ViT) all input and output grids must conform to a consistent processing shape, denoted as $H_{\text{proc}}, W_{\text{proc}}$.

This processing shape is dynamically computed for each individual puzzle. It is designed to accommodate the largest input or output grid observed within that puzzle's training examples, while also ensuring explicit divisibility by the Vision Transformer's patch size $P = 3$. Grids smaller than $H_{\text{proc}}, W_{\text{proc}}$ are zero-padded (filling empty space with colour 0, typically black); larger grids are cropped from their top-left corner to fit, although this is a rare occurrence given that ARC grid dimensions typically do not exceed 30×30 pixels. This rigorous sizing ensures uniform spatial dimensions for all grids processed by the ViT encoder, a critical step before patchification.

3.3.3 Paired Grid Concatenation

After one-hot encoding (as detailed in Section 3.3.1) and consistent sizing (Section 3.3.2), the final external preprocessing step before the data enters the Vision Transformer's internal architecture is combining the input and output grids from each training example.

The model we develop is builds on the idea of In-Context Learning (ICL). In the context of ARC, ICL enables a model to infer underlying transformation rules by explicitly leveraging demonstrations (the "context") provided alongside the input it needs to process. Our ICL framework provides relevant examples (Training Pair) directly within the input to guide the model's reasoning for a specific puzzle. While the model's weights are adapted through training for each unique ARC puzzle, the 'in-context' aspect refers to how the training pairs within that puzzle act as dynamic contextual cues that the model learns to interpret and apply in its forward pass. We consider ICL crucial for ARC due to the benchmark's emphasis on few-shot learning and generalisation from minimal examples, mirroring how humans infer rules from context.

To learn the transformation itself, the ViT encoder must see both the input grid $\mathbf{X}$ and its correct output grid $\mathbf{Y}$ at the same time. That is different from most machine learning approaches, where a model only ever sees $\mathbf{X}$ and tries to guess $\mathbf{Y}$. Here, we want the encoder to understand "how do we get from $\mathbf{X}$ to $\mathbf{Y}$?", so we give it both grids together. That way, it can directly pick up the rule linking them.

Therefore, the processed onehot($\mathbf{X}$) and onehot($\mathbf{Y}$) tensors from Section 3.3.1 are concatenated along their channel axis. This operation creates a single, unified 20-channel tensor, denoted as $\mathbf{Z}$:

$$\mathbf{Z} = \text{Concat}(\text{onehot}(\mathbf{X}), \text{onehot}(\mathbf{Y})) \in \{0,1\}^{20 \times H_{\text{proc}} \times W_{\text{proc}}} \tag{3.3.3}$$

This combined tensor $\mathbf{Z}$ serves as the direct input to the Vision Transformer encoder's initial tokenisation stage. By presenting the encoder with the joint state of the input grid $\mathbf{X}$ and its desired output grid $\mathbf{Y}$ from a training pair, we enable it to learn the implicit abstract mapping by observing their relationship. This capability is fundamental for the encoder to generate a meaningful "transformation embedding". This embedding is the objective of the encoder in our ICL framework, serving as the crucial context for the subsequent stages of the model.

3.4 Patchification and Token Construction

After the combined tensor $\mathbf{Z}$ is prepared, the Vision Transformer's internal processing pipeline begins by splitting this input into small, non-overlapping $P \times P$ patches. In our configuration, the patch size $P = 3$. The decision to form these patches from the concatenated $\mathbf{Z}$ tensor is crucial for obtaining the "transformation embedding" that serves as context for the model. This design ensures that the encoder can learn features derived from the joint spatial and colour information of the transformation, directly encompassing how elements in $\mathbf{X}$ relate to elements in $\mathbf{Y}$. This entire process at this stage is focused exclusively on learning from training pairs to generate the context; test inputs are handled in a later inference phase.

To ensure that $\mathbf{Z}$ can be perfectly subdivided into these patches, an *internal padding* step is applied if its dimensions H_{proc}, W_{proc} are not already exactly divisible by the patch size $P = 3$. While Chapter 2 defines the initial processing shape $H_{\text{proc}}, W_{\text{proc}}$ to largely align with architectural constraints (including ViT patch divisibility), this general shape is not guaranteed to be simultaneously perfectly divisible by ViT patch size Therefore, this internal padding acts as a stricter, final adjustment required specifically

by the ViT's patchification process. It extends the tensor's spatial dimensions (with zeros) to H' and W':

$$H' = P\lceil H_{\text{proc}}/P \rceil$$
$$W' = P\lceil W_{\text{proc}}/P \rceil$$

This guarantees perfect divisibility by $\mathbf{P}$ for the subsequent patch extraction, which is a strict requirement for the ViT's tokenisation process.

The padded $\mathbf{Z}$ tensor is then divided into N non-overlapping square patches of size $P \times P$ (3×3 pixels). Each patch inherently has 20 channels (from the concatenation of onehot($\mathbf{X}$) and onehot($\mathbf{Y}$)). The total number of patches N is determined by:

$$N = (H'/P) \times (W'/P)$$

Once these patches, denoted as $\mathbf{x}_p$ for $p = 1, \ldots, N$, are extracted, each is "flattened" into a one-dimensional vector. This involves unrolling the patch's 3D shape (channels, height, width) into a single 1D vector of length $20 \times P \times P$ (which is $20 \times 3 \times 3 = 180$ when $P = 3$). This flattening operation is formally denoted by $\text{vec}(\mathbf{x}_p)$.

Next, we turn this long pixel vector into a compact, learned embedding for the Transformer. This is achieved with a simple linear layer:

$$\mathbf{z}_p = W_{\text{patch}} \times \text{vec}(\mathbf{x}_p)$$

where $\mathbf{W}_{\text{patch}} \in \mathbf{R}^{D \times (20P^2)}$ is a trainable weight matrix and we set $D = 128$. $\mathbf{W}_{\text{patch}}$ projects the raw pixel values of each patch into a $\mathbf{D}$-dimensional patch embedding, $\mathbf{z}_p$. These patch embeddings are then collected into the final sequence $\mathbf{Z}_{\text{patch}}$:

$$\mathbf{Z}_{\text{patch}} = [\mathbf{z}_1; \mathbf{z}_2; \ldots; \mathbf{z_N}] \in \mathbb{R}^{N \times D} \tag{3.4.1}$$

Although this sequence lists every patch, it no longer carries any information about how those patches were arranged on the original two-dimensional grid. In other words, the Transformer cannot tell which patches were neighbors just by looking at their order in the sequence. To give each token a sense of *where* it came from, we will add learnable positional embeddings in the next Token Assembly step. These extra vectors re-encode the grid layout so the Transformer can reason about spatial relationships while it processes the sequence.

3.5 Transformer Architecture

After we convert the grid into a bunch of patch embeddings $\boldsymbol{Z}_{\text{patch}}$ (see Section 3.4), we pass them to the ViT. The Transformer layers let each patch to interact with every other patch, so the model can determine how the input turned into the output. In the end, it compresses all that information into one compact vector that summarises the transformation. It is important to note that while the original input $\mathbf{Z}$ (and thus its patches $\mathbf{x}_p$) contains the concatenated information of onehot($\mathbf{X}$) and onehot($\mathbf{Y}$) across 20 channels, the Transformer's layers learn to interpret the distinct patterns within these channel groups. Through its self-attention mechanism, the Transformer extracts features that capture the implicit mapping from $\mathbf{X}$ to $\mathbf{Y}$, guided by the overall objective of learning the transformation.

3.5.1 Class Token and Positional Embedding

The sequence of patch embeddings $\mathbf{Z}_{\text{patch}} \in \mathbb{R}^{N \times D}$ represents the visual content of the combined input $\mathbf{Z}$. However, to capture global information about the entire grid and to re-introduce spatial awareness lost during the flattening of patches, two additional types of tokens are added:

1. **Class Token ($\mathbf{z}_{\text{cls}}$):** A learnable vector, $\mathbf{z}_{\text{cls}} \in \mathbb{R}^D$, is prepended to the sequence of patch embeddings. This token serves as a special accumulator of global information across the entire grid. After processing through the Transformer's layers, the final state of this class token will embody the comprehensive transformation embedding.
2. **Positional Embeddings $\mathbf{P}$:** The Transformer's attention does not know about spatial order on its own, so we add a set of learnable positional embeddings $\mathbf{P} \in \mathbb{R}^{(N+1) \times D}$ to every token (including the class token). These embeddings carry information about where each patch came from on the original 2D grid, so the Transformer can understand how patches relate to one another in space.

The complete input sequence, $\boldsymbol{Z_0}$, is formed by concatenating the class token with the patch embeddings and then adding the positional embeddings:

$$\mathbf{Z}_0 = [\mathbf{z}_{\text{cls}}; \mathbf{z}_1; \ldots; \mathbf{z}_{\mathbf{N}}] + \mathbf{P} \tag{3.5.1}$$

where $\mathbf{z}_1, \ldots, \mathbf{z}_{\mathbf{N}}$ represent the patch embeddings from $\mathbf{Z}_{\text{patch}}$. This $\mathbf{Z}_0 \in \mathbb{R}^{(N+1) \times D}$ sequence is now ready for processing by the Transformer encoder layers.

3.5.2 Transformer Encoder Layers

The assembled sequence $\mathbf{Z}_0$ is passed through a stack of $L = 4$ identical Transformer encoder blocks. Each encoder block is designed to refine the token representations by allowing them to attend to each other, capturing complex dependencies. A standard Transformer encoder layer typically consists of two main sub-layers, each followed by Layer Normalisation (LN) and a residual connection:

1. **Multi-Head Self-Attention (MSA):** In this layer, each token looks at every other token and decides how much each one matters. Internally, we project each token into three vectors queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$ which are learned during training. By comparing queries and keys, the model determines which patches across the grid are most relevant to each other, capturing patterns that span the entire input. We then add this attended output back to the original tokens (a "residual" connection), giving:

   $$\mathbf{Z}' = \text{MSA}(\text{LN}(\mathbf{Z}_0)) + \mathbf{Z}_0 \tag{3.5.2}$$

   The attention scores are computed as

   $$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\,\mathbf{K}^T}{\sqrt{\mathbf{d}}}\right)\mathbf{V} \tag{3.5.3}$$

   where $d = 32$. This number comes from dividing the total embedding dimension D by the number of heads $h = 4$.

2. **Multi-Layer Perceptron (MLP):** After self-attention, each token is handled independently by a small feedforward network with two layers and a hidden size of 512, using GELU activations. This adds nonlinearity and gives the model flexibility to learn more complex transformations. Finally, we add the input of this MLP block back to its output (another residual connection), yielding:

   $$\mathbf{Z}_{\text{out}} = \text{MLP}(\text{LN}(\mathbf{Z}')) + \mathbf{Z}' \tag{3.5.4}$$

This sequential processing through multiple encoder blocks enables the Vision Transformer to build rich and abstract representations of the input, incorporating both local features and global contextual relationships.

3.5.3 Output: The Transformation Embedding

After the input has processed through all $L = 4$ encoder layers, we extract the final state of the "class" token. By this point, that token has paid attention to every other patch and collected information from across the entire input–output pair $\mathbf{Z}$. In other words, it holds a distilled summary of the transformation rule. We call this transformation embedding $\mathbf{E}$, and it is a 128-dimensional vector:

$$\mathbf{e} = \mathbf{Z}_{\text{out}}[0] \in \mathbb{R}^{128}. \tag{3.5.5}$$

where $\mathbf{Z}_{\text{out}}[0]$ refers to the class token's embedding from the final encoder layer. This vector acts as a unique signature summarising how the input grid $\mathbf{X}$ transforms into the output grid $\mathbf{Y}$ for a single training example.

For each puzzle we run the encoder on all K training pairs $\mathbf{X}^{(k)}, \mathbf{Y}^{(k)}$. That gives us one transformation embedding $\mathbf{e}^{(k)}$ per pair. We then stack those K vectors into a single "context" matrix:

$$\mathbf{E} = \left[\mathbf{e}^{(1)}; \mathbf{e}^{(2)}; \ldots; \mathbf{e}^{(K)}\right] \in \mathbb{R}^{K \times D}. \tag{3.5.6}$$

This context matrix $\mathbf{E}$ encapsulates the learned transformation rules from all training examples within a puzzle. It is this $\mathbf{E}$ matrix that will be passed to the next stage of our model to guide its predictions for unseen test inputs, as detailed in Chapter 4.

4. System Architecture: The U-Net Decoder

After the Vision Transformer (ViT), detailed in Chapter 3 processes the training pairs of a puzzle, it produces a set of transformation embeddings, $\mathbf{E} \in \mathbb{R}^{K \times 128}$ As described in Section 3.5.3, this context matrix $\mathbf{E}$ is a compact distilled summary of the abstract rules that map a puzzle's input grids $\mathbf{X}$ to its output grids $\mathbf{Y}$. The next stage of our framework is to apply these learned rules to a new, unseen test input $\mathbf{X}_{\text{test}}$. This crucial task is handled by a conditional U-Net decoder.

The U-Net is a convolutional neural network architecture originally designed for biomedical image segmentation, characterized by its symmetric encoder-decoder structure and skip connections Ronneberger et al. (2015). In the context of ARC-AGI, we adapt this architecture to transform an input grid $\mathbf{X}_{\text{test}}$ into an output grid $\hat{\mathbf{Y}}$, using the transformation embeddings $\mathbf{E}$ with a dimension of 128 as contextual guidance.

The core of our adaptation is the integration of Cross-Attention mechanisms at multiple stages of the decoder, a concept we will detail further in Section 4.2.6. This allows the model to continuously reference the learned transformation rules, weaving that abstract guidance into the spatial reconstruction process at every level of resolution. The overall flow through the decoder is illustrated in the right panel of the system architecture diagram (Figure 3.1).

4.1 Inputs to U-Net

The U-Net component is designed to accept two distinct inputs for its forward pass, each serving a specific purpose:

1. **The Target Input Grid:** This is the raw input grid for the current task, which corresponds to an unseen $\mathbf{X}_{\text{test}}$ during inference. Before being processed, this grid undergoes the same preprocessing steps detailed previously. It is first converted into a 10-channel tensor via one-hot encoding (as described in Section 3.3.1) and then padded or cropped to match the model's consistent processing dimensions, $H_{\text{proc}}, W_{\text{proc}}$ (as per Section 3.3.2).

$$\mathbf{X}_{\text{test}} \in \{0,1\}^{10 \times H' \times W'}$$

2. **The Transformation Context:** This is the context matrix $\mathbf{E}$, the final output from the ViT encoder (Equation 3.5.6). This tensor, with a shape of $K \times D$ (where K is the number of training examples for the puzzle and D is the embedding dimension of 128), contains the complete set of learned transformation rules.

$$\mathbf{E} = [\mathbf{e}^{(1)}; \mathbf{e}^{(2)}; \ldots; \mathbf{e}^{(K)}] \in \mathbb{R}^{K \times D}$$

4.2 U-Net Architecture

Once the U-Net receives its inputs (Section 4.1), the target grid is processed through a symmetric encoder-decoder architecture. The journey begins with an initial convolutional layer, followed by a contracting path that progressively extracts hierarchical features.

4.2.1 Initial Convolution

The 10-channel input tensor $\mathbf{X}_{\text{test}}$ first passes through an initial 3x3 convolutional layer. The purpose of this layer is to project the sparse one-hot representation into a dense, higher-dimensional feature space that is more suitable for the subsequent blocks. Specifically, it maps the 10 input channels to an initial feature dimension of 64 (`DECODER_INIT_CHANNELS`).

This convolution is applied with a padding of 1, which preserves the spatial dimensions $H_{\text{proc}}, W_{\text{proc}}$ of the input grid $\mathbf{X}_{\text{test}}$. The operation can be described as:

- **Input:** $\mathbf{X}_{\text{test}} \in \mathbb{R}^{10 \times H_{\text{proc}} \times W_{\text{proc}}}$
- **Weights:** $\mathbf{W}_1 \in \mathbb{R}^{64 \times 10 \times 3 \times 3}$
- **Output:** $\mathbf{F}_{\text{init}} = \text{ReLU}(\text{GroupNorm}(\text{Conv}_{3\times3}(\mathbf{X}_{\text{test}}, \mathbf{W}_1)))$

The resulting 64 feature maps encode low-level patterns, such as edges (colour transitions between adjacent pixels) and colour-specific activations, which are foundational for ARC-AGI's spatial transformations. For instance, a filter may learn to detect a horizontal edge where colour 3 (e.g., red) transitions to colour 0 (background), represented as a high activation in $\mathbf{F}_{\text{init}}[k, i, j]$ when the 3x3 neighbourhood matches this pattern Goodfellow et al. (2016).

4.2.2 Contracting Path (Encoder)

After the initial convolution, the feature maps, $\mathbf{F}_{\text{init}}$, are processed by the contracting path (encoder) of the U-Net. The path consists of Three levels, each designed to capture features at different scales. For each level we have a residual convolutional block followed by a max-pooling operation for downsampling.

4.2.3 The Feature Extractor: Residual Convolutional Blocks

The fundamental building block of our network is a residual convolutional block (`ConvBlock`). At each level of downsampling, this block's job is to analyse and refine the features at that particular scale. It consists of two standard convolution normalisation activation sequences. However, we made two critical design choices to tailor it to our task. First, instead of the common Batch Normalisation, we employ **Group Normalisation** Yuxin Wu (2018). This was a necessary decision, as our puzzle-specific training approach often uses an effective batch size of one, a scenario where Batch Normalisation is unstable. Group Normalisation ensures stable training regardless of batch size. Second, the block uses a **residual connection**, adding the input of the block to its output Kaiming (2015). This allows gradients to flow more easily during training and ensuring that fine-grained details from earlier in the block are not lost.

4.2.4 The Downsampling Operator: Max-Pooling

After the features at a given level have been refined, we use simple 2×2 max-pooling operation to move to the next level of downsampling. This operation halves the height and width of the feature map, forcing the subsequent layers to learn from a more global perspective.

4.2.5 The Journey Down the Path

The model's contracting path consists of three levels. The journey begins with the 64-channel features from the initial convolution:

- At **Level 1**, the features are first refined at their original resolution. The output, $\mathbf{F}_1 \in \mathbb{R}^{64 \times H_{\text{proc}} \times W_{\text{proc}}}$, is carefully set aside to be used later as a **skip connection**. The feature map is then downsampled.
- At **Level 2**, the model now looks at a spatially coarser grid. To capture more complex patterns, the feature dimension is doubled to 128. After refinement, this new map, $\mathbf{F}_2 \in \mathbb{R}^{128 \times (H_{\text{proc}}/2) \times (W_{\text{proc}}/2)}$, is also saved as a skip connection before the next downsampling step.
- This process culminates at **Level 3**, where the feature complexity is doubled one last time to 256 channels. The resulting map, $\mathbf{F}_3 \in \mathbb{R}^{256 \times (H_{\text{proc}}/4) \times (W_{\text{proc}}/4)}$, is saved.

After a final max-pooling step, we are left with a small but information-rich feature map of size $\mathbb{R}^{256 \times (H_{\text{proc}}/8) \times (W_{\text{proc}}/8)}$. This tensor, which represents the most abstract understanding of the input grid, is then passed to the U-Net's bottleneck. Meanwhile, the stored skip connections $\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3$ hold the high-resolution spatial information that will be essential for accurately reconstructing the output grid on the way back up.

4.2.6 The Bottleneck: The Meeting of "What" and "How"

At the bottom of the U-shape, we have the bottleneck, the conceptual heart of the decoder. Up to this moment, the U-Net has been focused on a single task: analysing the input grid $\mathbf{X}_{\text{test}}$ to understand *what* it contains, breaking it down into a rich, abstract map of features. The ViT encoder has been working on a different task: analysing the demonstration pairs to understand *how* the puzzle works, distilling this logic into the context matrix $\mathbf{E}$. The bottleneck is where these two streams of understanding finally meet.

This process, illustrated in our system architecture diagram (Figure 3.1), is not a simple concatenation but a multi-stage procedure. The bottleneck receives the feature map from the final level of the contracting path with dimensions of $\mathbb{R}^{256 \times (H_{\text{proc}}/8) \times (W_{\text{proc}}/8)}$ and processes it in three steps:

1. **Feature Deepening:** First, the input feature map is processed by a `ConvBlock`, which doubles the feature dimension from 256 to 512. This step serves to create a richer, more abstract representation of the input grid.
2. **Context Injection via Cross-Attention:** This is the central operation of the bottleneck. The deepened, 512 channel feature map is now passed to a **Cross-Attention** block. This mechanism allows the spatial features of the input grid (which act as "queries") to attend to the transformation embeddings stored in the context matrix $\mathbf{E}$ **(where each embedding is the final output state of the ViT encoder's '[CLS]' token, as detailed in Section 3.5.1)** (which provide the "keys" and "values") Vaswani et al. (2017). This enables each region of the feature map to selectively integrate the most relevant puzzle rule, effectively modulating the features based on the puzzle's specific logic.
3. **Final Refinement:** After the features have been infused with the contextual rules, a final `ConvBlock` processes them. This block maps the feature dimension back down from 512 to 256 channels, finalising the representation for the expanding path.

The output of the bottleneck is a feature map, $\mathbf{F}_{\text{bottle}} \in \mathbb{R}^{256 \times (H_{\text{proc}}/8) \times (W_{\text{proc}}/8)}$, that now is a high-level plan for the final grid. This context-rich feature map is now ready to begin its journey up the expanding

path, which will translate these high-level instructions into the final, pixel-perfect output.

4.2.7 The Expanding Path: Reconstructing the Grid with Guidance

The purpose of this path is to reverse the process of the contracting path: it takes the abstract, low-resolution feature $\mathbf{F}_{\text{bottle}}$ and translates it into a full-resolution, pixel-perfect output grid. This reconstruction is a careful, step-by-step process that happens over three upsampling levels, mirroring the three levels of the contracting path.

Each level in the expanding path follows four step procedure, designed to combine high-level semantic guidance with low-level spatial detail:

1. **Upsampling:** The process starts with a 2×2 transposed convolution (`ConvTranspose2d`). This operation doubles the height and width of the feature map while halving the number of feature channels, taking the first step towards rebuilding the spatial resolution.
2. **Re-injecting Detail via Skip Connections:** Next, the upsampled feature map is concatenated with the corresponding high-resolution feature map that was saved from the contracting path (e.g $\mathbf{F}_3$). This **skip connection** is the defining feature of the U-Net architecture. It acts as an information highway, re-introducing the spatial details that were inevitably lost during the downsampling process. If there is a minor size mismatch between the two feature maps, we use interpolation to ensure they align perfectly.
3. **Applying Contextual Guidance via Cross-Attention:** This is the most sophisticated step in the process. Now that the high-level plan and the low-level details have been merged, a **Cross-Attention** block is applied. This is a critical distinction of our architecture: instead of applying the rules only once, we re-apply them at every scale of the reconstruction. This allows the model to use the puzzle's rules (from the context matrix $\mathbf{E}$) to guide how the high-resolution details are integrated. For example, the rules can help decide exactly how a pattern should be rendered at this specific resolution, ensuring global consistency.
4. **Final Feature Refinement:** Finally, the combined, context-aware feature map is passed through one last `ConvBlock`. This block refines the features at the current resolution, preparing them for the next level of upsampling.

This four-step repeats for each of the three levels, progressively increasing the resolution and reducing the feature complexity (256 $\rightarrow$ 128 $\rightarrow$ 64 channels) until we have a full-resolution feature map ready for the final output.

4.3 Final Output Generation

The reconstruction process is completed in two final steps to produce the output grid:

1. **Final Logit Projection:** The 64 channel feature map from the last level of the expanding path is passed through a final 1×1 convolution. This layer acts as a simple projection, mapping the 64 rich features for each pixel down to just 10 output *logits* one for each of the 10 possible ARC colours.
$$\hat{\mathbf{Y}}^{\text{logits}} = \text{Conv}_{1\times1}(\mathbf{F}_{\text{final}}, \mathbf{W}_F) \quad \in \mathbb{R}^{10\times H_{\text{proc}}\times W_{\text{proc}}}$$
2. **colour Selection:** To produce the final grid, we perform `argmax` operation over the colour dimension for each pixel. This simply selects the colour with the highest logit score, or confidence,

at each position.

$$\hat{\mathbf{Y}}[i,j] = \arg\max_{c \in \{0,\dots,9\}} \hat{\mathbf{Y}}^{\mathsf{logits}}[c,i,j] \tag{4.3.1}$$

During inference, we take one last step. We crop the predicted grid, which has the padded dimensions $(H_{\mathsf{proc}}, W_{\mathsf{proc}})$, to match the target output shape.

5. Methodology

The preceding chapters have detailed the two core components of our system: the Vision Transformer (ViT) encoder and the conditional U-Net decoder. Together, they form an architecture designed to learn and apply abstract rules from context.

This chapter shifts the focus from architectural theory to practical application. We will now describe the end-to-end methodology used to solve a given ARC puzzle. This includes our specific strategy for data preparation, the task-specific training pipeline, the loss function used for optimisation, and the final inference process for generating predictions on unseen test grids.

5.1 Data Preparation

For any given ARC puzzle, the first step is to prepare its data for the task-specific model. The process begins by taking in the puzzle's complete set of training and test inputs. Each training instance provides an input grid $\mathbf{X}$ and a corresponding ground-truth output grid $\mathbf{Y}$, while each test instance provides only the input grid $\mathbf{X}_{\text{test}}$.

To make these grids suitable for neural network processing, we transform them into a consistent shape and format. This involves two main stages:

5.1.1 Grid Representation

First, every grid is converted into a 10-channel binary tensor using one-hot encoding, a process detailed in Section 3.3.1. This ensures that the discrete colour values are not misinterpreted as having an ordinal relationship.

5.1.2 Consistent Sizing

As ARC grids vary in size, each input and output grid for a given puzzle is padded or cropped to a common processing shape, $H_{\text{proc}}, W_{\text{proc}}$. The calculation proceeds as follows:

1. **Base Dimensions:** We start with the maximum height and width observed across all training grids for the specific puzzle.
2. **U-Net Constraint:** To ensure the U-Net's downsampling path (detailed in Section 4.2.2) does not collapse the feature maps to zero, the dimensions must be at least 2^L, where L is the number of downsampling levels.
3. **ViT Constraint:** To allow for patchification, the dimensions must be at least equal to the ViT's patch size (from Section 3.4).
4. **Consolidation:** A base dimension, $H_{\text{base}}, W_{\text{base}}$ is determined by taking the maximum of the training-set sizes and the minimum sizes required by the U-Net and ViT.
5. **Final Sizing:** This base dimension is then rounded up to the nearest multiple required for both U-Net and ViT divisibility, yielding the final processing shape $H_{\text{proc}}, W_{\text{proc}}$.

This task-specific sizing ensures that we preserve as much information as possible from the original grids while adhering to the architectural requirements of our model. The precise padding logic used to conform the grids to this shape is described in Section 3.3.2.

5.2 The Task-Specific Training Pipeline

Once the data for a puzzle is prepared, we train a dedicated model using a strategy built upon the principles of **In-Context Learning (ICL)** Tom B. Brown (2020). In the ICL paradigm, a model learns to perform a task by conditioning on a few input-output examples provided directly within its context. Our framework implements a specialised version of this through a **leave-one-out** training objective. In this setup, the model is explicitly trained to solve for one training example by using the other training examples as context. This process is repeated for a fixed number of training iterations, allowing the model's parameters to gradually absorb the puzzle's underlying transformation rules.

At each step of the training process, the model performs a full pass over the K available training pairs. For each training pair $(\mathbf{X}^{(k)}, \mathbf{Y}^{(k)})$ chosen as the target, the procedure illustrated in our system architecture diagram (Figure 3.1) is done as follows:

5.2.1 Transformation Encoding Stage

The first stage within the training loop is to generate the guiding context. For a given target pair $\mathbf{X}^{(k)}, \mathbf{Y}^{(k)}$, the model must distil the puzzle's rules from the remaining $(K-1)$ demonstration pairs. The process for generating a single transformation embedding $\mathbf{e}^{(j)}$ from a context pair $\mathbf{X}^{(j)}, \mathbf{Y}^{(j)}$ is detailed below and illustrated in the "Encoder Input" panel of the system architecture diagram (Figure 3.1).

Input Construction for the ViT Encoder First, the input grid $\mathbf{X}^{(j)}$ and output grid $\mathbf{Y}^{(j)}$ are pre-processed according to the steps in Section 5.1. The resulting one-hot tensors, $\text{onehot}(\mathbf{X}^{(j)})$ and $\text{onehot}(\mathbf{Y}^{(j)})$, are then concatenated along their channel axis to form a single 20-channel tensor, which we denote as $\mathbf{Z}$ (Equation 3.3.3).

$$\mathbf{Z} = \text{concat}(\text{onehot}(\mathbf{X}^{(j)}), \text{onehot}(\mathbf{Y}^{(j)})) \in \{0,1\}^{20 \times H_{\text{proc}} \times W_{\text{proc}}}$$

This explicit pairing of the "before" and "after" states is a cornerstone of our approach, as it enables the encoder to learn the features of the transformation itself.

ViT Encoding Process

This combined tensor $\mathbf{Z}$ is then passed to the Vision Transformer (ViT) Encoder, whose high-level structure was introduced in Chapter 3. The encoding process involves transforming the grid into a sequence of tokens and processing them through Transformer layers.

1. **Patchification and Projection:** The input tensor is first divided into a sequence of non-overlapping $P \times P$ image patches, where our patch size $P = 3$. These patches are then flattened and linearly projected into 128-dimensional "patch embeddings," $(\mathbf{z}_p)$, forming the initial sequence of tokens $\mathbf{Z}_{\text{patch}}$ as described in Section 3.4.

   $$\mathbf{z}_p = \mathbf{W}_{\text{patch}} \cdot \text{vec}(\mathbf{x}_p)$$

   where $\mathbf{x}_p$ is a patch, $\text{vec}(\cdot)$ is the flattening operation, and $\mathbf{W}_{\text{patch}}$ is the learned projection matrix. The resulting sequence is $\mathbf{Z}_{\text{patch}} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N]$.

2. **Token Assembly:** As detailed in Section 3.5.1, a learnable `[CLS]` token embedding,$\mathbf{z}_{\text{cls}}$, is prepended to this sequence. This special token acts as a global accumulator for the transformation representation. Learnable positional embeddings, $\mathbf{P}$, are then added to every token to

re-introduce the spatial awareness that was lost when the grid was broken into a sequence of patches. The final input sequence to the Transformer layers, $\mathbf{Z}_0$, is given by:

$$\mathbf{Z}_0 = [\mathbf{z}_{\text{cls}}; \mathbf{z}_1; \dots; \mathbf{z}_N] + \mathbf{P} \tag{5.2.1}$$

3. **Transformer Encoder Layers:** The sequence $\mathbf{Z}_0$ is processed by a stack of $L = 4$ standard Transformer Encoder layers. As described in Chapter 3.5, each layer contains a Multi-Head Self-Attention (MSA) sub-layer and a Feed-Forward Network (FFN) sub-layer. The core operation is the Scaled Dot-Product Attention, defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{5.2.2}$$

This mechanism allows the `[CLS]` token to aggregate information from all patch tokens, effectively creating a summary of the entire input-output grid pair.

Output Transformation Embedding

After the sequence has been processed by all $L = 4$ Transformer layers, the final hidden state corresponding to the **[CLS]** token is extracted. This 128-dimensional vector, $\mathbf{e}^{(j)}$, serves as the compact "transformation embedding" for the pair $\mathbf{X}^{(j)}, \mathbf{Y}^{(j)}$, as defined in Equation 3.5.5.

$$\mathbf{e}^{(j)} = (\text{ViTEncoder}(\mathbf{Z}_0))_0$$

Finally, these $(K-1)$ individual transformation embeddings are stacked together to form the complete context matrix $\mathbf{E}_{\neg k}$, which represents the collective knowledge of the puzzle's rules available for this specific learning step.

5.2.2 Conditional Output Generation Stage

The decoder takes two inputs: the pre-processed target input grid $\mathbf{X}^{(k)}$, and the context matrix $\mathbf{E}_{\neg k}$, which was generated from the demonstration pairs. The target grid $\mathbf{X}^{(k)}$ is processed through the U-Net architecture, whose components are detailed in Chapter 4, with the context $\mathbf{E}_{\neg k}$ being injected at multiple stages to guide the generation (see Figure 3.1. The entire flow is detailed below.

Initial Convolution and Contracting Path

First, the target grid $\mathbf{X}^{(k)}$ passes through the initial 3×3 convolutional layer (Section 4.2.1) and the three levels of the contracting path (Section 4.2.2). This process extracts a hierarchy of feature maps at decreasing spatial resolutions, which we can denote as $\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3$. These feature maps are saved to be used later as skip connections. The final output of this stage is the most abstract feature map, $\mathbf{F}_3 \in \mathbf{R}^{256 \times (H_{\text{proc}}/8) \times (W_{\text{proc}}/8)}$, which is then passed to the bottleneck.

The Bottleneck

At the bottleneck (Section 4.2.6), the abstract feature map $\mathbf{F}_3$ is first deepened to 512 channels by a `ConvBlock`. Then, in the central operation of the decoder, a cross-attention mechanism fuses these features with the context matrix $\mathbf{E}_{\neg k}$. This allows the features representing the content of $\mathbf{X}^{(k)}$ to be

modulated by the learned rules of the puzzle. A final `ConvBlock` processes the result, producing the context-aware feature map $\mathbf{F}_{\text{bottle}}$, which now serves as the high-level blueprint for the output.

The Expanding Path: Reconstructing the Grid with Guidance

The model now starts to reconstruct the output grid by proceeding up the expanding path (Section 4.2.7). This process happens over three upsampling levels, each of which performs a precise four step procedure:

1. **Upsampling:** A 2×2 transposed convolution doubles the spatial dimensions of the feature map from the previous level.
2. **Skip Connection:** The upsampled features are concatenated with the corresponding high-resolution feature map $\mathbf{F}_3$ that was saved from the contracting path. This re-injects the fine-grained spatial details of the original input.
3. **Contextual Guidance:** A cross-attention block is applied to the combined features. This is a critical step where the context matrix $\mathbf{E}_{\neg k}$ is used once again to guide the reconstruction, ensuring the puzzle's rules are respected at every spatial scale.
4. **Feature Refinement:** A final `ConvBlock` processes the guided features, preparing them for the next upsampling level.

This cycle repeats until a full-resolution feature map, $\mathbf{F}_{\text{final}} \in \mathbb{R}^{64 \times H_{\text{proc}} \times W_{\text{proc}}}$, is produced.

Final Output Generation

The final feature map $\mathbf{F}_{\text{final}}$ is passed through a 1×1 convolutional layer which projects the 64 channels down to 10 channels, one for each colour class. This produces the final grid of predicted logits, $\hat{\mathbf{Y}}^{\text{logits}}$, for the target output.

$$\hat{\mathbf{Y}}^{\text{logits}} = \text{Conv}_{1\times1}(\mathbf{F}_{\text{final}}, \mathbf{W}_F) \quad \in \mathbf{R}^{10 \times H_{\text{proc}} \times W_{\text{proc}}} \tag{5.2.3}$$

This logit grid is now ready to be compared against the ground-truth grid $\mathbf{Y}^{(k)}$ to calculate the loss for this training step.

5.3 Loss Function and Optimisation

The final step in the leave-one-out training loop is to quantify the model's prediction error and use it to update the network's parameters. This is achieved through a carefully chosen loss function and a standard Optimisation algorithm.

5.3.1 Loss Function: Weighted Cross-Entropy

The training objective is to minimise the difference between the decoder's predicted logits, $\hat{\mathbf{Y}}^{\text{logits}}$, and the one-hot encoded ground-truth output grid, $\mathbf{Y}^{(k)}$. For this multi-class classification task (where each pixel is a prediction across 10 colour classes), the standard choice is the Cross-Entropy Loss.

However, ARC puzzles often exhibit a severe class imbalance. A grid might be 95% background colour (black, colour 0) with only a few pixels of other, more critical colours (see Figure 6.1. A standard

cross-entropy loss would be dominated by the background, encouraging the model to become very good at predicting black pixels while ignoring the rare colours that define the puzzle's core logic.

To counteract this, we employ a **Weighted Cross-Entropy Loss**. This approach assigns a higher penalty for misclassifying rare colours, forcing the model to pay more attention to them. The weight $\mathbf{w}_c$ for each colour c, is calculated once per puzzle based on the colour frequencies in its training outputs:

1. First, we count the total number of pixels for each colour across all training output grids of the puzzle.
2. The raw weight for each colour is then calculated as the inverse of its frequency, which gives higher weight to rarer colours. We add a small epsilon for numerical stability:
$$w_c^{\text{raw}} = \frac{1}{\text{frequency}(c) + 10^{-6}}$$
3. To prevent the background colour from dominating, its weight w_0 is explicitly set to a fixed, lower value of 0.3 (`BACKGROUND_WEIGHT`).
4. Finally, to prevent extreme values, all weights are clamped to a range of $[0.1, 10.0]$.

This final weight vector $\mathbf{w}$ is then used in the cross-entropy loss function. To build up to our final formula, we first define how the model's raw output logits are converted into probabilities using the Softmax function.

Let $p_{ij,c}$ be the predicted probability of the pixel at position i, j being colour c. This is calculated from the vector of 10 logits for that pixel, $\hat{\mathbf{y}}_{ij}$, as follows:

$$p_{\mathbf{ij,c}} = \text{softmax}(\hat{\mathbf{y}}_{ij})_c = \frac{\exp(\hat{\mathbf{y}}_{ij,c})}{\sum_{\mathbf{c}'=0}^{9} \exp(\hat{y}_{ij,c'})} \tag{5.3.1}$$

where $\hat{y}_{ij,c}$ is the predicted logit for colour c at pixel i, j.

With this probability, we can now define the standard Cross-Entropy (CE) loss for that single pixel, which simply measures the negative log probability of the true colour:

$$\text{CE}(\hat{\mathbf{y}}_{ij}, y_{ij}) = -\log(p_{ij,y_{ij}}) \tag{5.3.2}$$

where y_{ij} is the single integer representing the true colour label.

Our final loss function, L, is the mean of this CE loss calculated over all pixels in the grid, but with our special weights applied. Given the predicted logits $\hat{\mathbf{Y}}$ and the ground-truth grid $\mathbf{Y}^{(k)}$, the loss is:

$$\mathcal{L}(\hat{Y}, Y^{(k)}) = -\frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} w_{y_{ij}^{(k)}} \times \text{CE}(\hat{\mathbf{y}}_{ij}, y_{ij}^{(k)}) \tag{5.3.3}$$

Here, $y_{ij}^{(k)}$ is the true colour label for the pixel i, j from the ground-truth grid $\mathbf{Y}^{(k)}$. The weight $w_{y_{ij}^{(k)}}$ is the special weight we calculated for that true colour. By multiplying the standard CE loss by this weight, we ensure that mistakes on rare colours are penalized more heavily.

5.3.2 Parameter Optimisation

The calculated loss is used to update all learnable parameters in both the ViT encoder and the U-Net decoder via backpropagation. We use the **AdamW** optimiser, a robust and widely used variant of the Adam optimiser that improves weight decay regularisation (Loshchilov and Hutter, 2017).

- **Learning Rate:** We use a starting learning rate of 3×10^{-4}.
- **Learning Rate Schedule:** To improve convergence, the learning rate is not kept constant. We employ a **Cosine Annealing Schedule**, which smoothly anneals the learning rate from its initial value down to a minimum value over the course of the training iterations. This allows for larger updates at the beginning of training and finer, more stable adjustments as the model converges.
- **Gradient Clipping:** To prevent exploding gradients, which can destabilise training, we apply gradient clipping. After the gradients are computed but before the optimiser step, their L_2 norm is clipped to a maximum value of 1.0.
- **Early Stopping:** To prevent overfitting and save computational resources, we employ an early stopping mechanism with a patience of 10. If the validation loss does not improve for 10 consecutive evaluation intervals, the training run is halted.

Through this carefully configured optimisation process, the entire network learns to both generate and interpret transformation embeddings effectively.

5.4 The Inference Pipeline: Solving Unseen Tasks

After the puzzle-specific model has been fully trained using the pipeline described in Section 5.2, the final stage is to use it for inference. This process involves generating predictions for the one or more unseen test input grids, $\mathbf{X}_{\text{test}}$, provided with the puzzle. The inference pipeline consists of two main phases: generating a definitive context from all available knowledge, and then using that context to make predictions.

5.4.1 Final Context Generation

During training, we used a leave-one-out approach where the context for a given target was formed from the other $(K-1)$ examples. For inference, however, we want the model to have access to all available information to make the best possible prediction.

So, the first step is to create a single, definitive context matrix for the puzzle. To do this, we pass **all** $\mathbf{K}$ of the training pairs $\mathbf{X}^{(i)}, \mathbf{Y}^{(i)}$ through the trained ViT encoder. This produces K transformation embeddings, which are stacked into a final context matrix, $\mathbf{E}$

$$\mathbf{E} = \left[\mathbf{e}^{(1)}; \mathbf{e}^{(2)}; \ldots; \mathbf{e}^{(K)}\right] \in \mathbb{R}^{K \times D}$$

This context matrix, which represents the complete learned knowledge of the puzzle's rules, is generated only once and is then reused for predicting all test grids for that puzzle.

5.4.2 Test Grid Prediction

With the final context matrix prepared, the model then iterates through each provided test input grid, $\mathbf{X}_{\text{test}}$. For each test grid, the following steps are performed:

1. **Preprocessing:** The test grid $\mathbf{X}_{\text{test}}$ undergoes the same data preparation steps outlined in Section 5.1. It is converted to its one-hot tensor representation and padded to the consistent processing shape, $H_{\text{proc}}, W_{\text{proc}}$, that was used during training.
2. **Conditional Generation:** The pre-processed test grid is passed as input to the trained U-Net decoder, along with the final context matrix $\mathbf{E}$. The decoder processes the grid through its full architecture, guided at multiple scales by the contextual information, to produce a final grid of logits, $\hat{\mathbf{Y}}^{\text{logits}}$.
3. **Colour Selection:** An `argmax` operation is performed across the colour channel of the logit grid. This selects the colour with the highest confidence score for each pixel, producing the final predicted output grid, $\hat{\mathbf{Y}}$, at the padded resolution (Equation 4.3.1).

$$\hat{\mathbf{Y}}[i, j] = \arg \max_{c \in \{0,\dots,9\}} \hat{\mathbf{Y}}^{\text{logits}}[c, i, j]$$

5.4.3 Final Output Cropping

The output grid $\hat{\mathbf{Y}}$ produced by the decoder has the padded dimensions $H_{\text{proc}}, W_{\text{proc}}$. However, a successful submission to the ARC benchmark requires that the predicted grid has the exact dimensions of the hidden ground truth.

To achieve this, we employ a content-aware cropping mechanism. This procedure operates on the assumption that the model has learned to produce the core output content surrounded by a uniform background colour (black, colour 0), which fills the rest of the padded canvas. To determine the final shape, we compute the minimal bounding box that encapsulates all non-background pixels in the predicted grid $\hat{\mathbf{Y}}$. The grid is then cropped to this bounding box. This approach allows the model's own output to dynamically define its final dimensions, making the sizing process an integral part of the prediction.

6. Results

This chapter discusses the results from applying our system to a number of ARC puzzles. We will start with an in-depth walkthrough of a single, illustrative puzzle to build an intuition for how the model learns and reasons. Then we will present the quantitative results across the full benchmark, providing a clear measure of the system's overall effectiveness. Finally, we will analyse common patterns of both success and failure to better understand the cognitive strengths and weaknesses of the approach.

6.1 Example Puzzle

To illustrate the model's operational process, we will examine a concrete example. We return to ARC puzzle **0ca9ddb6**, which was the running example in Chapter 2. For convenience, we reproduce Figure 2.1 in Figure 6.1 below. By walking through this puzzle again, we can see how the system learns the underlying pattern and then applies that pattern to new grids, giving a more intuitive feel for its in-context learning skills and overall performance.

Figure 6.1 showcases the training examples and test input for puzzle **0ca9ddb6**. Observing the training pairs, a clear transformation rule can be inferred:

- **Red squares:** These are covered by 4 diagonal yellow squares, placed directly over the same position.
- **Blue squares:** These are covered by 4 orange cross squares, also placed directly over the same position.
- **Cyan squares:** These squares stay exactly the same.

This rule is applied consistently across all training pairs. The transformation depends entirely on the input colour of each square and applies the same logic universally.

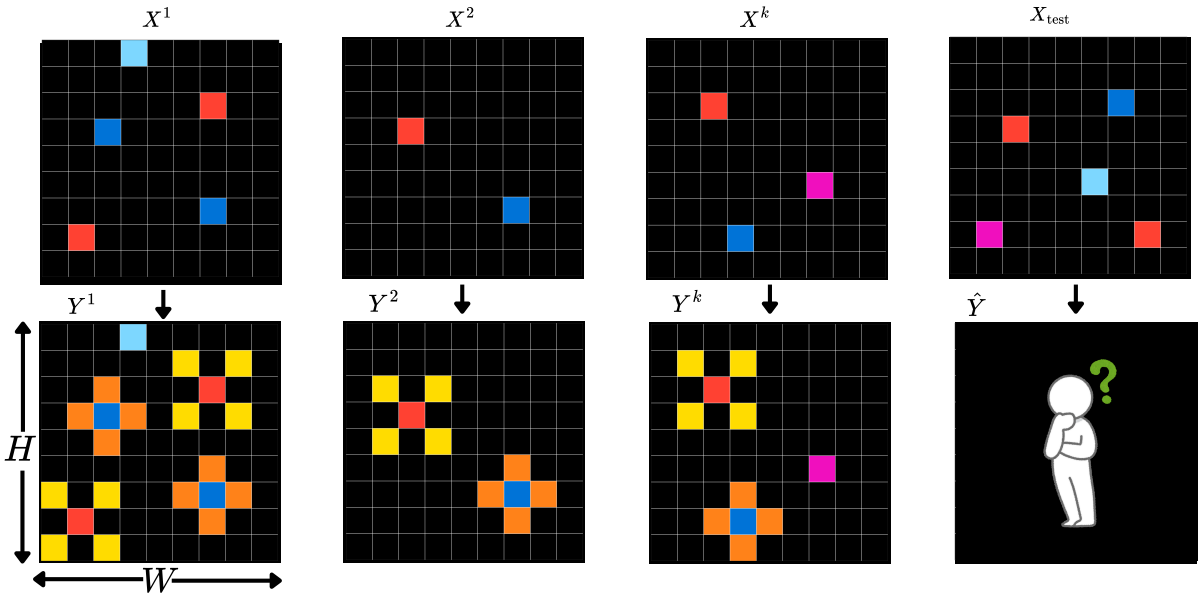


Figure 6.1: This figure illustrates the structure of an ARC-AGI-2 task for challenge **0ca9ddb6**. This puzzle is generally considered straightforward for humans. The ease for a human stems from a simple, step-by-step analytical process. First, upon observing the initial training pair, a human would form a hypothesis about the colour-based rules: that red squares are covered by a yellow diagonal pattern, blue squares are covered by a orange cross pattern, and cyan squares remain unchanged. Next, this set of rules would be verified against the second training pair to confirm their consistency. Finally, having validated the transformation logic, the human can confidently apply it to the unseen test input to generate the correct output. This entire process is facilitated by what the thesis refers to as "core knowledge priors" the innate human ability to perceive discrete objects, generalise patterns, and formulate simple conditional rules without extensive training, a key challenge for AI systems.

6.1.1 Training Loss

With this understanding of the underlying transformation, we can now see how the model applies this logic. The training process for our model, as detailed in Chapter 5, employs a leave-one-out In-Context Learning objective, where the goal is to minimise the weighted cross-entropy loss detailed in Equation 5.3.3.

To understand how this in practice, let us consider one specific step of the training loop for the puzzle shown in Figure 6.1. For this step, imagine we designate the first training pair the one labelled $(\mathbf{X}^1)$ and $(\mathbf{Y}^1)$ as our **target**. The goal for our model is to correctly predict the output $(\mathbf{Y}^1)$ after only seeing the input $(\mathbf{X}^1)$.

To do this, the model is given the **other two** training pairs as **context**. It will look at the second pair ($\mathbf{X}^2$ and $\mathbf{Y}^2$) and the third pair ($\mathbf{X}^k$ and $\mathbf{Y}^k$). The Vision Transformer (ViT) encoder processes these two context pairs to determine the rule: that red squares get yellow diagonals and blue squares get orange crosses. It distils this learned rule into a single context vector, $(\mathbf{E})$, as detailed in Section 3.5.3.

Finally, this context vector $(\mathbf{E})$ is fed alongside the target input $\mathbf{X}^1$ into the U-Net decoder (Chapter

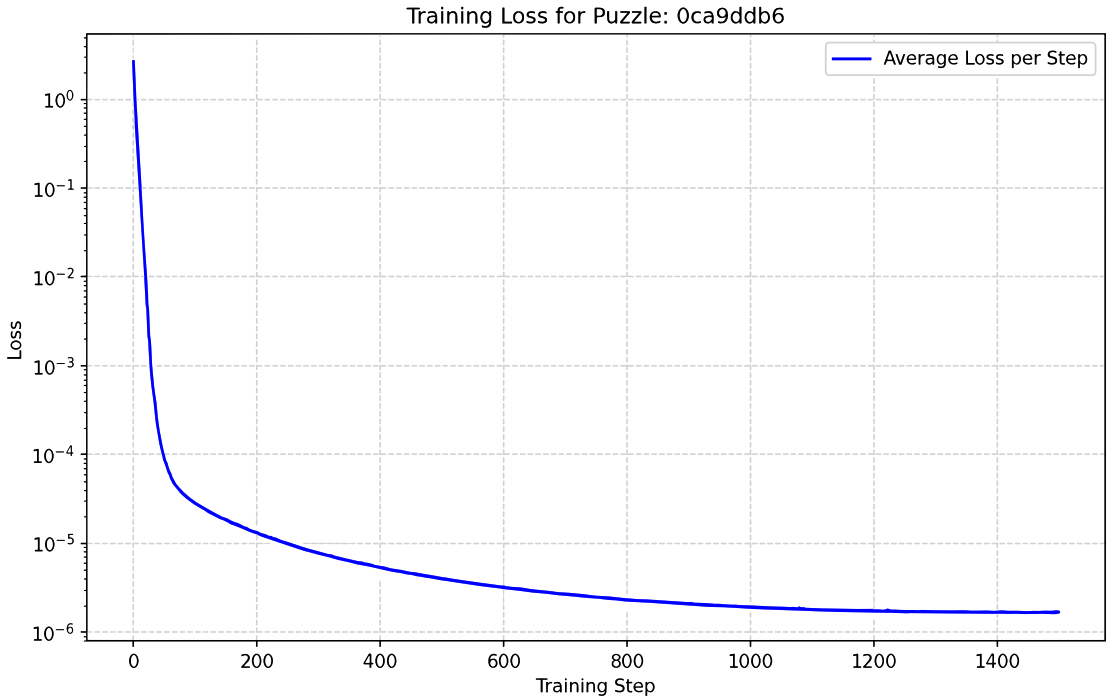


Figure 6.2: The training loss from Equation 5.3.3 for puzzle **0ca9ddb6** over 1500 training iterations.

4). The decoder, now armed with the rule provided by the context vector, attempts to apply it to $\mathbf{X}^1$ to generate a final prediction. The difference between this prediction and the true target output $\mathbf{Y}^1$ is then used to calculate the loss (Equation 5.3.3) and update the model's weights, as conceptually illustrated in our system architecture (Figure 3.1).

Our encoder-decoder system is trained **from scratch** specifically for each individual ARC puzzle. For puzzle **0ca9ddb6**, we train the model for 1500 iterations (gradient descent steps), carefully pushing its parameters to minimise the weighted cross-entropy loss (Section 5.3.1) between its predictions and the actual ground-truth outputs. In the training loss shown in Figure 6.2, there is a steady decline over the iterations.

Beyond the overall training loss, it is insightful to examine the model's confidence for individual pixel predictions on a test example. After the 1500 training iterations, we can visualize the raw outputs, known as logits, and their corresponding softmax probabilities for selected pixels from the first test input of puzzle **0ca9ddb6** (Figure 6.4).

6.1.2 Analysis of Top-4 Predictions

Figure 6.3 shows the top four most likely output grids based on the model's single set of predicted logits. The "Top 1" prediction is the model's final prediction, generated by taking the colour with the highest probability (the argmax 4.3.1) at each pixel location. The "Top 2" prediction is generated by taking the colour with the **second-highest** probability at each pixel, and so on. Each prediction grid is generated from the same initial set of logits.

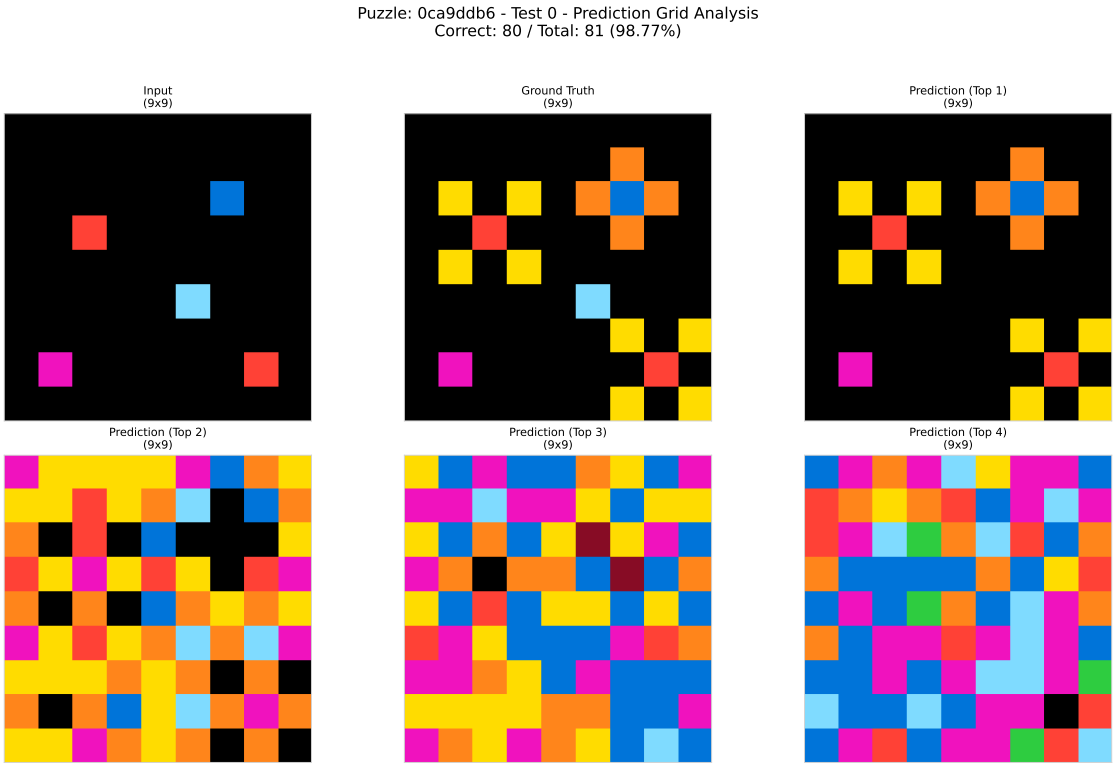


Figure 6.3: The top four most likely predictions for the first test input of puzzle **0ca9ddb6**. The **Ground Truth** shows the correct solution, while the **Input** is what the model was given. The **Prediction (Top 1)** grid is the final answer, created by choosing the colour with the highest probability at each pixel. The **Prediction (Top 2)** is made from the colours with the second-highest probability, and so on. The near-perfect match of the Top 1 prediction (98.77% pixel accuracy) demonstrates the model's high confidence in the correct solution.

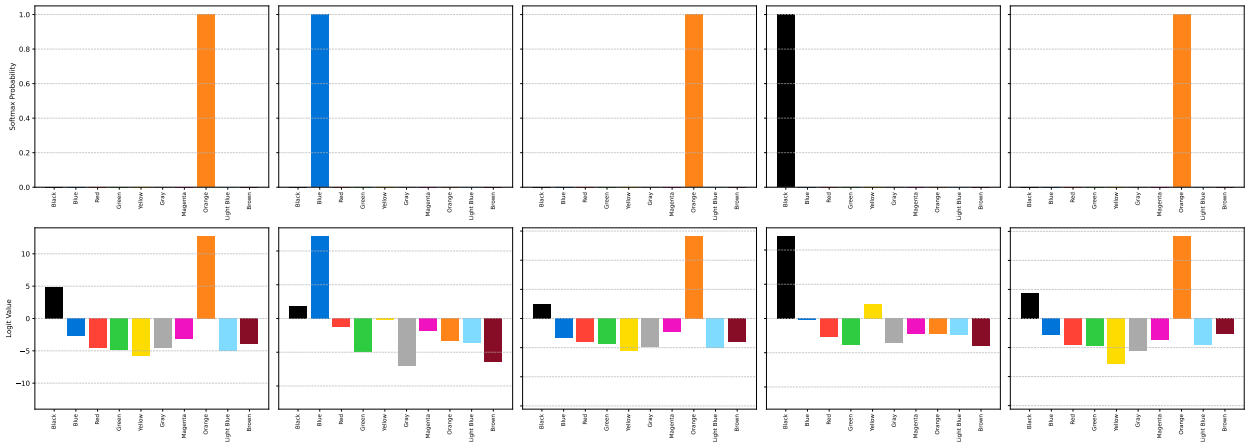


Figure 6.4: A detailed look at the model's output for several key pixels from puzzle **0ca9ddb6** (see Figure 6.1 for the full puzzle). For each pixel shown, the top chart displays the raw output **logits** ($\hat{\mathbf{y}}_{ij}$), which represent the model's raw confidence. The bottom chart shows the final softmax probabilities ($\mathbf{p}_{ij,c}$), which are derived from the logits using the Softmax function defined in Equation 5.3.1.

6.1.3 Logits: Pre-softmax Colour Scores per Pixel

Figure 6.4 provides a close look at the model's decision-making process, showing the raw scores it generates for several key pixels. Logits are the unnormalised, raw scores assigned by the model to each possible colour for a given pixel, where higher values indicate stronger confidence. Softmax probabilities then convert these logits into a probability distribution over the ten colours, summing to one, directly representing the model's belief for each colour at that pixel location. For instance, in background areas like pixel (0,0) or (4,4) (which are input colour 0 and ground truth output colour 0), the model correctly assigns a very high logit to colour 0 (black), leading to a near 1.0 softmax probability for black. This indicates strong confidence in maintaining the background. More tellingly, consider pixel (8,8). Here, the input is colour 0, but the ground truth output is colour 4 (yellow). The model successfully predicts colour 4, evident by the significantly high logit value for colour 4 compared to all other colours, resulting in a near 1.0 softmax probability for yellow. This particular example highlights the model's capacity to learn intricate, context-dependent transformations that involve changing even background pixels to a specific target colour based on learned patterns from the training set, not just simple colour-to-colour mappings. While the complete test grid for puzzle **0ca9ddb6** contains 81 pixels (you can find the full Figure in Appendix), Figure 6.4 selectively presents only a few representative pixels to clearly illustrate the model's decision-making process without overwhelming the reader with a full grid of probability distributions.

6.1.4 Per-Pixel colour Predictions

After looking at individual pixels, Figure 6.5 gives us a fascinating overview: per-colour probability heatmaps for the same test input. Think of it like this: for each colour (from 0 to 9), you get a map of the whole grid showing how confident the model is that that specific colour should appear at each pixel. Dark spots on a heatmap mean the model is pretty sure about that colour for that pixel, while lighter spots mean it is less convinced. This gives us a full picture of the model's "confidence landscape" for every possible output colour across the entire grid. For example, if you look at the heatmap for colour 0 (black), you will see it is dark in most background areas, showing the model confidently leaves the background untouched unless a transformation applies. Now, here is where it gets interesting: the

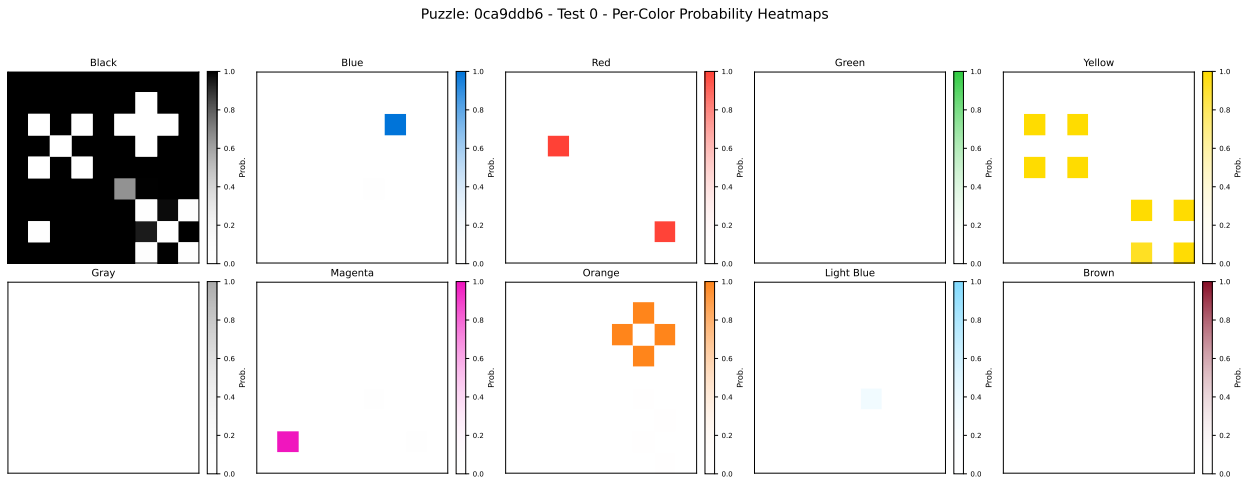


Figure 6.5: Each pixel in an Abstract Reasoning Corpus (ARC) puzzle can take one of $C = 10$ colour values. The predictions for puzzle **0ca9ddb6** are shown here for each of the $C = 10$ colour channels. The original puzzle is in Figure 6.1. The figure shows the top prediction out of four different predictions, which are illustrated in Figure 6.3.

heatmap for colour 4 (yellow) lights up exactly where those red squares from the training examples were covered by diagonals yellow. Similarly, the heatmap for colour 7 (orange) highlights the cross patterns that came from the blue squares. By seeing these maps, you can spot how the model is simultaneously thinking about all 10 colours for every single pixel. When it is time for the final prediction (which you can see in Figure 6.3), the model simply picks the colour with the highest probability at each pixel, essentially deciding which colour wins out on each map. This global view of its probabilities highlights the model's knack for not just learning specific transformation rules, but applying them intelligently across the whole grid, a powerful example of its in-context learning in action.

6.2 Performance Analysis

We evaluate our solver with the puzzle-level accuracy, the fraction of test grids whose outputs are an exact match, as formally defined in **Equation 2.3.2**. In addition, we report the official Kaggle Pass@2 score, which applies the same exact-match criterion but allows two attempts per test grid, detailed in Section 2.3.

Figure 6.6 shows the distribution of these puzzle-level accuracies across the 833 puzzles in the ARC-AGI-2 public training set. The histogram reveals a strong positive skew, indicating that the model performs well on a large portion of the tasks. Notably, 71 puzzles (8.5%) are solved with 100% accuracy, and a total of 306 puzzles (36.7%) achieve an accuracy of 90% or higher, demonstrating the model's ability to fully or almost-fully generalise on a significant subset of the benchmark.

6.2.1 Accuracy

The comprehensive performance statistics are summarised in **Table 6.1**.

The model solves **78 puzzles perfectly**, scores at least 90% accuracy on 311 puzzles, and averages 73.8% accuracy over 833 from training tasks. These numbers confirm that the system generalises well beyond the few training demonstrations provided for each puzzle.

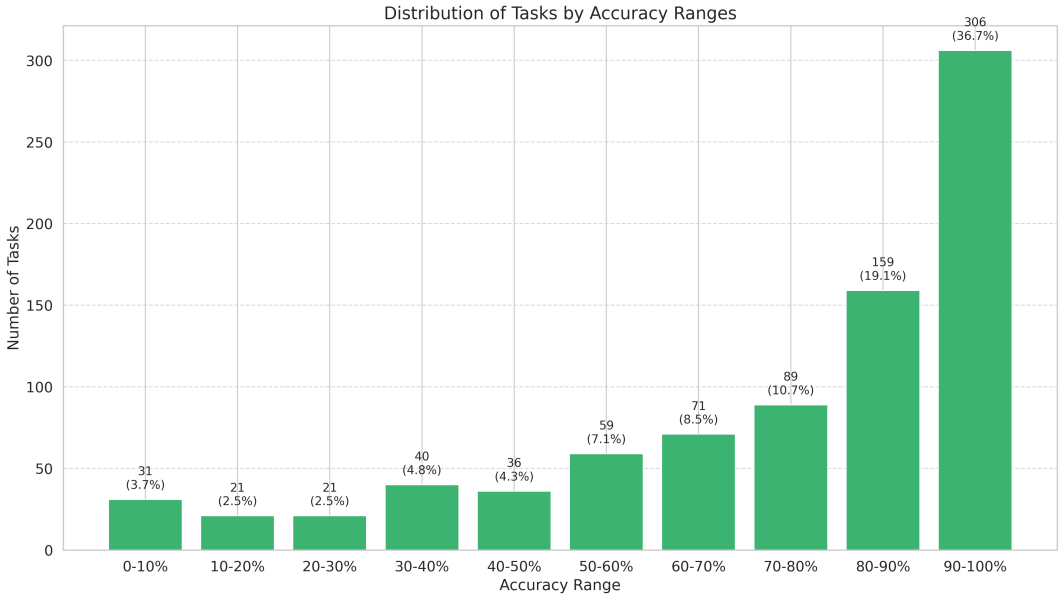


Figure 6.6: The accuracies on each of the 833 puzzle in **ARC-AGI-2 public training set**, as a histogram in 10% accuracy increments. The puzzle accuracy is measured according to the exact-match criterion defined in Equation 2.3.2. If a puzzle is a 9×9 puzzle, an accuracy of 90-100% means that out of 81 pixels, at least 72 pixels are predicted correctly. Out of 833 puzzles, 71 puzzles were predicted with 100% accuracy.

Metric	Value
Tasks evaluated	833
Solved tasks (perfect)	78
Accuracy ≥ 0.90	311
Mean puzzle accuracy	0.738
Kaggle Pass@2 score	0.830

Table 6.1: Overall statistics across 833 ARC-AGI tasks. The **Kaggle Pass@2 score of 0.83** is the accuracy of our best submission achieved on the private test split

Task ID	Accuracy	Task ID	Accuracy
00576224	0.583	05a7bcf2	0.587
007bbfb7	0.346	05f2a901	0.945
009d5c81	0.867	0607ce86	0.968
00d62c1b	0.932	0692e18c	0.827
00dbd492	0.880	06df4c85	0.904
017c7c7b	0.889	070dd51e	0.868
025d127b	0.980	08ed6ac7	0.691
03560426	0.930	09629e4f	0.818
045e512c	0.834	09c534e7	0.849
0520fde7	0.444	0a1d4ef5	0.083
	⋮		
0ca9ddb6	0.987	Figure 2.1's task	

Table 6.2: The puzzle-level accuracy for the first 20 puzzles from the 833 puzzle of the training set, sorted alphabetically by Task ID. This highlights the wide range of performance, from near perfect scores (e.g., **025d127b**) to tasks on which our model achieved lower performance. The final row highlights the score for puzzle **0ca9ddb6**, which is used as a running example throughout this thesis.

6.2.2 Per-puzzle Breakdown

While the overall statistics provide a high-level summary, discovering the performance on individual puzzles reveals the wide diversity in task difficulty. **Table 6.2** presents the accuracy scores for the first 20 puzzles from the 833 puzzle of the training set, sorted alphabetically by Task ID. This sample illustrates the typical spectrum of outcomes, from near perfect solutions to more challenging tasks. A comprehensive list of results for all 833 puzzles can be found in Appendix A.2.

6.3 Qualitative Analysis: Areas of Success

While the overall performance metrics in Section 6.2 show the overall effectiveness of our model, they do not tell the full story of its reasoning capabilities. To better understand that, this section presents a qualitative analysis of the model's behaviour on specific puzzle archetypes. We examine categories of transformations where the model consistently succeeds and, just as importantly, where it fails. This analysis aims to characterise the abstract reasoning skills our architecture learns and to identify its current limitations, thereby providing clear directions for future research.

6.3.1 Frequency Analysis

A key success is the model's ability to solve tasks based on object frequency. In puzzle **39a8645d** (Figure 6.7), the transformation rule is to identify the single shape that repeats most often in a given colour, and then to construct the output using only instances of that specific shape. Successfully solving this puzzle requires grouping objects by both shape and colour, counting the members of each group, and filtering the output based on the highest count. The model's perfect execution of this task is a significant achievement, as it demonstrates that it has learned an effective strategy for cardinality based reasoning without any explicit counting module.

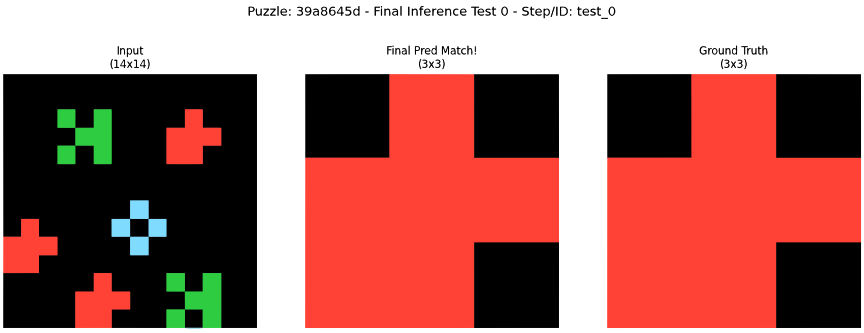


Figure 6.7: Successful prediction for puzzle **39a8645d**. The task requires identifying the most frequent shape and colour combination from the input (here, the red shapes). The model correctly performs this frequency analysis, isolating the most common pattern and using it to construct the output while correctly filtering out all other, less frequent shapes.

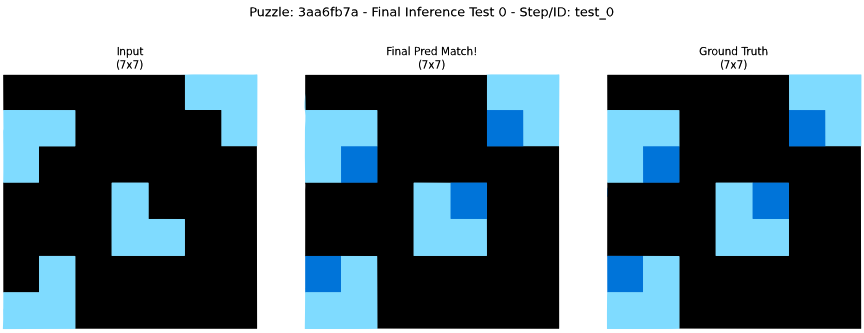


Figure 6.8: Successful prediction for puzzle **3aa6fb7a**. The task is to recognize that the input 'L' shapes are incomplete 2x2 squares and complete them. The model correctly performs this geometric inference, learning the local spatial correlation to fill the missing corner of each 'L' shape with a dark blue pixel to form a full square.

6.3.2 Implicit Shape Reconstruction

The architecture demonstrates a strong ability to learn mappings from partial or incomplete patterns to complete geometric shapes. For puzzle **3aa6fb7a** (Figure 6.8), the transformation consists of recognizing L-shaped pixel motifs as incomplete 2x2 squares and generating the missing pixel to complete them with dark blue colour.

This is achieved through the learning of strong local correlations. The model's patch-based and convolutional processing learns that the specific input feature of an "L" motif has a near 100% correlation with a full 2x2 square in the ground-truth output across the training examples. It therefore learns a high fidelity transformation for this specific local feature, effectively performing a geometric reconstruction based on the learned training data distribution, rather than any explicit knowledge of squares.

6.3.3 Conditional Feature Templating

A more advanced capability is the learning of highly conditional, context-dependent transformations. In puzzle **321b1fc6** (Figure 6.9), the model must apply the complex colour texture from one distinct input pattern (the "source") onto the spatial locations of several other, simpler patterns (the "targets"), but

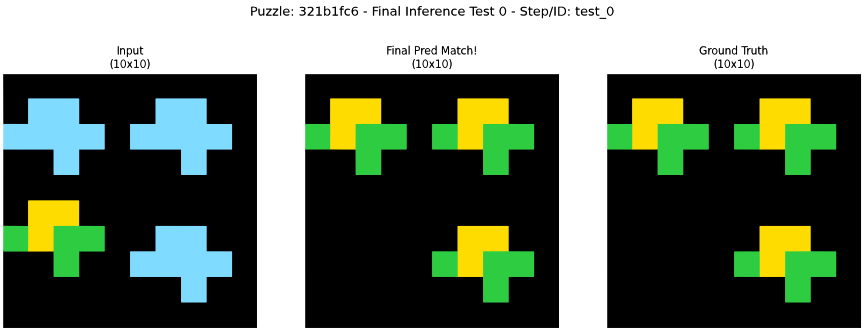


Figure 6.9: Successful prediction for puzzle **321b1fc6**. This task requires a conditional templating operation where the colour texture from a "source" object is applied to all "target" objects of a specific shape. The model correctly identifies the multi-coloured source pattern and the light-blue "+" target canvases, and successfully transfers the texture, demonstrating an ability to reason about object roles.

they most be with the same shape.

This demonstrates that the context embedding learned by the ViT acts as a powerful modulator for the U-Net decoder. The decoder learns to not just perform a single transformation, but to use the context to execute a conditional operation: identify the features and spatial coordinates of the target patterns, identify the textural features of the source pattern, and then transfer those textural features to the target locations. This ability to learn a feature templating and application process is a clear step towards solving more hierarchical and compositional reasoning tasks.

6.4 Qualitative Analysis: Error Analysis

Despite its successes, the architecture struggles with tasks that require reasoning beyond direct pattern application, particularly those involving topological integrity, hierarchical structures, or spatially dependent conditional logic.

6.4.1 Failure in Preserving Topological Integrity

The model's internal representation of shapes appears to be feature-based and lacks a robust concept of structural integrity. This is demonstrated in puzzle **810b9b61** (Figure 6.10), where the task is to recolour the boundary of closed loops. While the model correctly identifies the target shapes, its generated output for one of the loops is topologically flawed it breaks the continuous boundary of the shape it is transforming. This suggests that while the model learns to associate input and output features, it cannot guarantee the preservation of fundamental properties like closure and connectivity.

6.4.2 Failure in Spatially Grounded Conditional Logic

The model is unable to learn transformations where the rule is specified in one region of the input grid and must be applied to objects in another meaning the model must look at the first tiny object inside the gray area and apply the colour to the first outer object and same for others. In puzzle **1da012fc** (Figure 6.11), the recolouring transformation is defined by a "legend" inside a distinct gray box. The model fails completely, indicating that it cannot differentiate between pixels that specify a rule versus pixels that represent the data to be transformed. It processes the input as a single, flat field of features, preventing it from learning this higher-order, spatially dependent conditional logic.

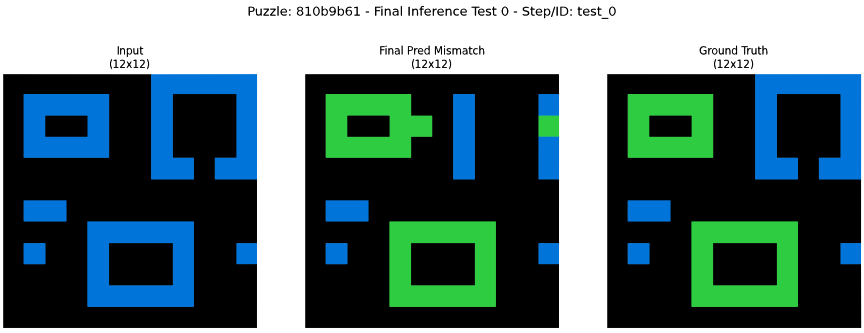


Figure 6.10: Failure on puzzle **810b9b61**. The objective is to find all closed loops and recolour their boundaries. The model correctly identifies the target loops but fails to preserve their structural integrity. In the prediction for the top-right loop, the model breaks the continuous boundary, showing a weakness in reasoning about object topology.

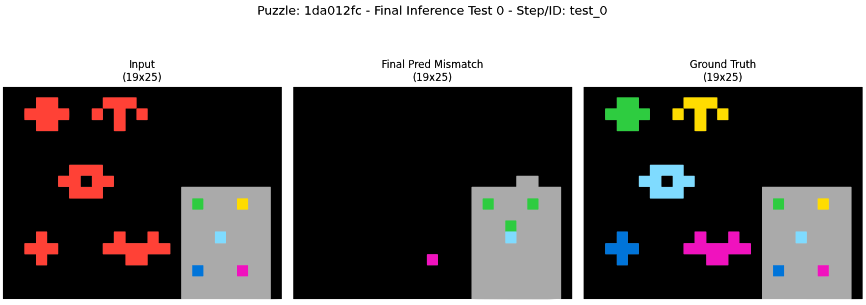


Figure 6.11: Failure on puzzle **1da012fc**. The transformation rule is specified in the gray "legend" box, where the position of a coloured pixel dictates the new colour for a corresponding target object. The model fails to learn this spatially-grounded conditional logic. Its output is chaotic, indicating it could not differentiate between the rule-specifying region and the data region, treating the entire grid as a single scene.

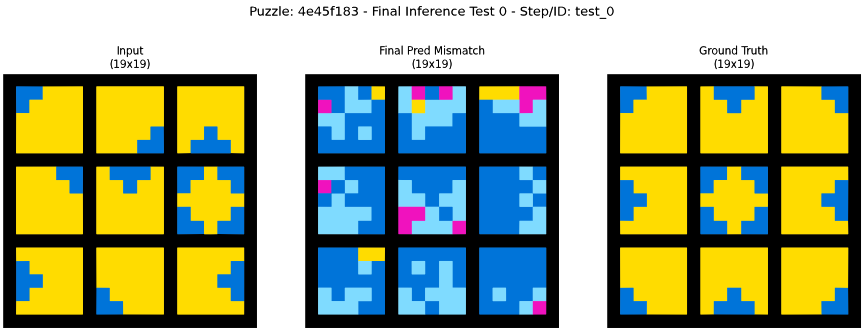


Figure 6.12: Failure on puzzle **4e45f183**. This task requires a global jigsaw-style reassembly. The input presents a grid of cells, each containing a fragment of a larger, coherent pattern. The correct transformation is to rearrange these cells as indivisible tiles to form the organised final image. The model fails because it completely misinterprets the nature of the task. Instead of learning to reorder the cells, it defaults to applying a local, pixel-level transformation *within* the static boundaries of each cell. This highlights a critical failure to abstract from local feature transformation to global object manipulation, demonstrating that the model cannot infer the global context required to switch its reasoning strategy.

6.4.3 Inability to Process Hierarchical Grid Structures

The architecture proves to be limited when faced with hierarchically structured tasks. Puzzle **4e45f183** (Figure 6.12) is composed of a grid of cells, where the transformation must be applied independently within each cell, based on the global understanding. The model fails to recognize this crucial structure. As a result, its prediction is chaotic.

7. Conclusion

This thesis began with the long-standing quest to build a thinking machine, a challenge that has defined the field of Artificial Intelligence since its inception. We took up this challenge by confronting what we termed the "core deficit" of modern AI: the chasm between its power in statistical pattern matching and its weakness in abstraction and reasoning. The Abstraction and Reasoning Corpus (ARC) served as our crucible, a benchmark designed not to test rote learning, but to probe for the fluid intelligence that allows humans to solve novel problems on the fly.

We explored a fundamentally different path from the prevailing symbolic and LLM-driven approaches, which search for an explicit programme. Instead, we investigated whether a purely neural system, operating directly on pixels, could learn to reason implicitly. Our proposed solution was a context-aware neural architecture, a two-stage system composed of a Vision Transformer encoder and a U-Net decoder. In a deliberate departure from common practice, we trained a fresh, specialised model for each puzzle from scratch. This forces the architecture to derive the logic exclusively from the examples provided, it is a direct implementation of the In-Context Learning paradigm.

The results of this exploration are both encouraging and sobering. On one hand, the successes are undeniable. The model's ability to solve a significant portion of the ARC tasks, including those requiring frequency analysis and implicit shape reconstruction, demonstrates that a neural network can indeed learn to perform abstract reasoning without an explicit symbolic engine. It learned to complete patterns, and to apply conditional rules, all by observing the relationships in the data.

On the other hand, the failures. The model's inability to preserve topological integrity, its failure on tasks with spatially grounded conditional logic, and its chaotic predictions on hierarchical grid structures reveal the sharp edges of its capabilities. It showed a shallow grasp of objecthood, breaking the continuous boundaries of a shape it was meant to transform. It could not differentiate between pixels that represented data and pixels that specified a rule, treating a legend as just more features in a flat field. These shortcomings confirm that our architecture, while context-aware, still lacks a robust, internal model of the world.

This work, therefore, serves as both a proof of concept and a map for future inquiry. It validates the pursuit of purely neural, context-driven reasoning systems while clearly illuminating the missing cognitive faculties a deeper understanding of objects, hierarchy, and causality that must be instilled to bridge the final gap towards AGI. The path is long, but the results presented in this thesis suggest it is a viable and promising one.

Acknowledgements

I want to acknowledge the African Institute for Mathematical Sciences (AIMS) and its funders. Their support was foundational, providing the environment and resources that made this research possible.

This thesis felt like a marathon, and I couldn't have reached the finish line without the guidance of my supervisors. I am deeply grateful to Prof. Ulrich Paquet, whose supervision went far beyond the high-level. His dedication to every detail, his clarity of thought, and his line-by-line engagement with this work challenged me to push further at every step. His mentorship was instrumental not just in shaping this thesis, but in shaping how I think.

I am equally grateful to my co-supervisor, Jaron Cohen. His meticulous attention to detail also delivered line by line was a constant source of learning.

I also need to thank our Academic Director, Claire David. Her leadership creates an environment at AIMS where students can genuinely focus on their work, and for that, we are all lucky.

Finally, to my family and friends, thank you for your unwavering support.

References

Akyürek, E., Kanodia, A., Gothe, J. G. M., Foster, D. J., and Andreas, J. The surprising effectiveness of test-time training for abstract reasoning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/html/2411.07279v1.

Alex Krizhevsky, G. E. H., Ilya Sutskever. Imagenet classification with deep convolutional neural networks. In *ImageNet Classification with Deep Convolutional Neural Networks*, 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

ARC Prize, Inc. 2024 progress. Official Prize Website, 2024. URL https://arcprize.org/blog/2024-progress-arc-agi-pub.

ARC Prize, Inc. Arc prize 2025 competition details. Official Prize Website, 2025. URL https://arcprize.org/competition.

Berman, J. How i got a record 53.6% on arc-agi. Substack, 2024. URL https://jeremyberman.substack.com/p/how-i-got-a-record-536-on-arc-agi.

Brachman, R. J. 'I lied about the trees' or, defaults and definitions in knowledge representation. *The AI Magazine*, 1985. URL https://onlinelibrary.wiley.com/doi/epdf/10.1609/aimag.v6i3.490.

Brenden M. Lake, J. B. T. S. J. G., Tomer D. Ullman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 2016. URL https://arxiv.org/abs/1604.00289.

Cattell, R. B. Theory of fluid and crystallized intelligence: A critical experiment. *Journal of Educational Psychology*, 1963. doi: 10.1037/h0046743.

Chollet, F. The abstraction and reasoning corpus. *arXiv preprint arXiv:1911.01547*, 2019. URL https://arxiv.org/abs/1911.01547.

Colmerauer, A. and Roussel, P. The birth of Prolog. In *History of Programming Languages*. ACM Press, 1996. URL https://dl.acm.org/doi/10.1145/234286.1057820.

d'Avila Garcez, A. S. and Lamb, L. C. Neurosymbolic ai: The 3rd wave. *Artificial Intelligence Review*, 2023. doi: 10.1007/s10462-023-10499-9.

Deng, J., Dong, W., Socher, R., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/pdf/2010.11929.

Dreyfus, H. L. *What Computers Can't Do: The Limits of Artificial Intelligence*. Harper & Row, 1972.

Ernst, G. W. and Newell, A. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, 1969. URL https://search.worldcat.org/title/1057908548.

Feigenbaum, E. A. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, 1977. URL https://stacks.stanford.edu/file/druid:bg342cm2034/bg342cm2034.pdf.

Feldman, J. A. and Ballard, D. H. Connectionist models and their properties. *Cognitive Science*, 1982.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hassabis, D., Kumaran, D., Summerfield, C., and Botvinick, M. Neuroscience-inspired artificial intelligence. *Neuron*, 2017. doi: 10.1016/j.neuron.2017.06.011.

Haugeland, J. *Artificial Intelligence: The Very Idea*. MIT Press, 1985. URL https://direct.mit.edu/books/book/4347/Artificial-IntelligenceThe-Very-Idea.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 1997.

icecuber. 1st place solution write-up. Kaggle, 2020. URL https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154597.

Kaiming, S. J., Zhang. Deep residual learning for image recognition. In *Deep Residual Learning for Image Recognition*, 2015. URL https://arxiv.org/pdf/1512.03385.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. In *Neural Computation*, 1989.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature*, 2015. URL https://www.nature.com/articles/nature14539.

Li, W., Xu, Y., Sanner, S., and Khalil, E. B. Tackling the abstraction and reasoning corpus with vision transformers: the importance of 2d representation, positions, and objects. In *ICLR 2025 Conference*, 2025. URL https://openreview.net/forum?id=0gOQeSHNX1.

Li, W.-D., Ellis, K., Mushfig, A., Solar-Lezama, A., and Tavares, Z. Combining induction and transduction for abstract reasoning. In *Proceedings of the 2024 Conference on Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2411.02272.

Lighthill, J. *Artificial Intelligence: A General Survey*. Science Research Council, 1973.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Marcus, G. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018. URL https://arxiv.org/abs/1801.00631.

Marcus, G. The next decade in ai: Four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177*, 2020.

McCarthy, J., Minsky, M. L., Rochester, N., and Shannon, C. E. A proposal for the dartmouth summer research project on artificial intelligence. *AI Magazine*, 2006. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1904.

McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 1943.

Minsky, M. and Papert, S. A. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

Newell, A. and Simon, H. A. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 1976. URL https://dl.acm.org/doi/10.1145/360018.360022.

Pearl, J. and Mackenzie, D. *The Book of Why: The New Science of Cause and Effect*. Basic Books, 2018.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI 2015)*, 2015. doi: 10.1007/978-3-319-24574-4_28. URL https://arxiv.org/abs/1505.04597.

Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 1958.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 1986.

Russell, S. J. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020. URL https://aima.cs.berkeley.edu/.

Shortliffe, E. H. *Computer-Based Medical Consultations: MYCIN*. Elsevier, 1976. URL https://www.researchgate.net/publication/238720842_Computer-based_medical_consultations_MYCIN.

Silver, D., Huang, A., Maddison, C. J., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016. URL https://www.researchgate.net/publication/292074166_Mastering_the_game_of_Go_with_deep_neural_networks_and_tree_search.

Spelke, E. S. and Kinzler, K. D. Core knowledge. *Developmental Science*, 2007. doi: 10.1111/j.1467-7687.2007.00569.x.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., and Goodman, N. D. How to grow a mind: Statistics, structure, and abstraction. *Science*, 2011. URL https://cocosci.princeton.edu/tom/papers/LabPublications/GrowMind.pdf.

Tom B. Brown, N. R. M. S. J. K., Benjamin Mann. Language models are few-shot learners. In *Language Models are Few-Shot Learners*, 2020. URL https://arxiv.org/pdf/2005.14165.

Turing, A. Computing machinery and intelligence. *Mind*, 1950. URL https://courses.cs.umbc.edu/471/papers/turing.pdf.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 2017. doi: 10.48550/arXiv.1706.03762. URL https://arxiv.org/abs/1706.03762.

von Rueden, L. et al. Informed machine learning – a taxonomy and survey of integrating prior knowledge into learning systems. *IEEE Transactions on Knowledge and Data Engineering*, 2023. doi: 10.1109/TKDE.2021.3079836.

Werbos, P. J. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, 1974.

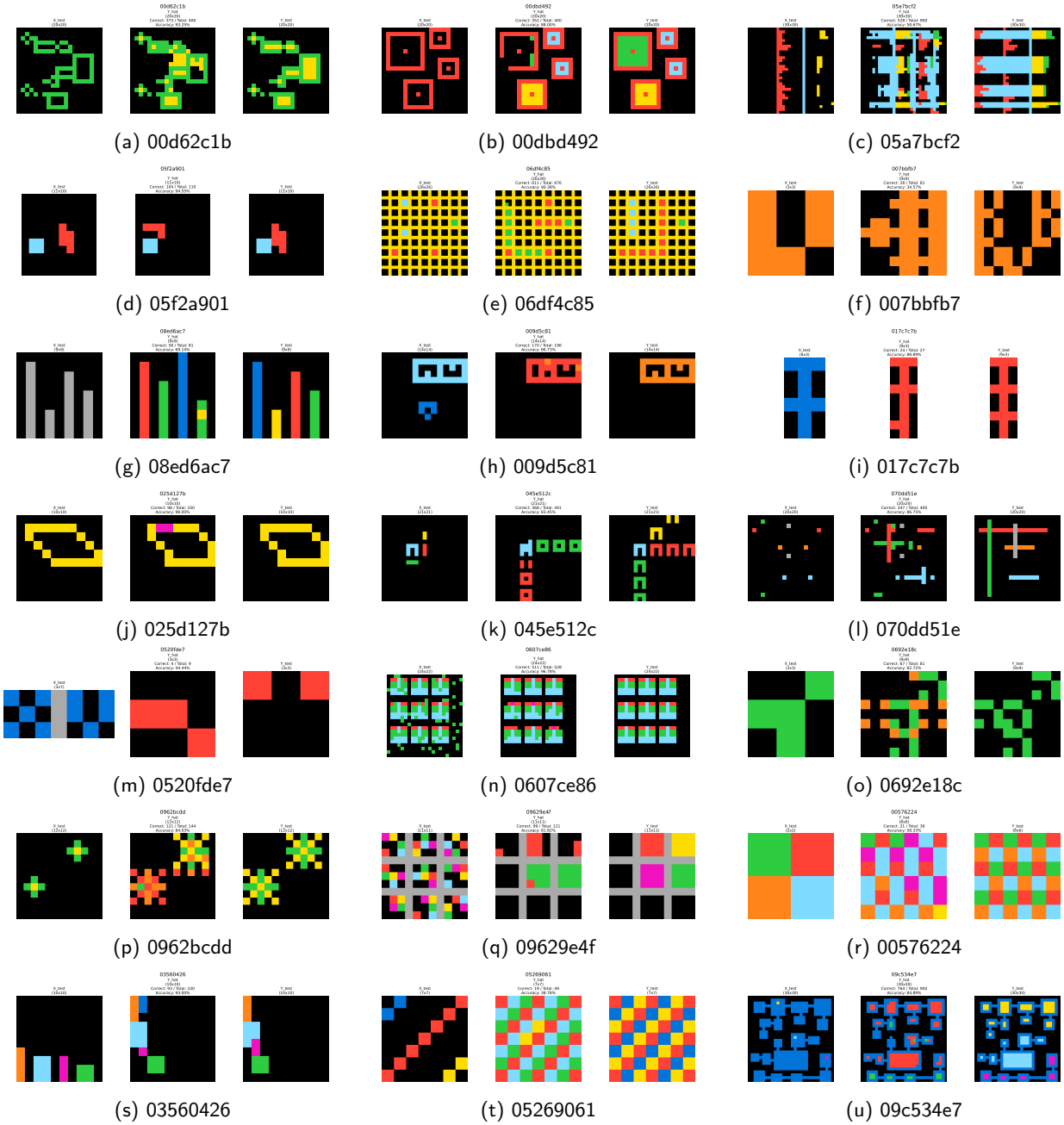
Xu, Y., Khalil, E. B., and Sanner, S. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the Neuro Causal and Symbolic AI Workshop at NeurIPS 2022*, 2022. URL https://ncsi.cause-lab.net/pdf/nCSI_4.pdf.

Yuxin Wu, K. H. Group normalization. In *Group Normalization*, 2018. URL https://arxiv.org/pdf/1803.08494.

Appendix A. Detailed Results

A.1 Example Predictions

This section presents 20 unique predictions generated by our model for various test inputs of ARC puzzles.



(a) 00d62c1b (b) 00dbd492 (c) 05a7bcf2

(d) 05f2a901 (e) 06df4c85 (f) 007bbfb7

(g) 08ed6ac7 (h) 009d5c81 (i) 017c7c7b

(j) 025d127b (k) 045e512c (l) 070dd51e

(m) 0520fde7 (n) 0607ce86 (o) 0692e18c

(p) 0962bcdd (q) 09629e4f (r) 00576224

(s) 03560426 (t) 05269061 (u) 09c534e7

Figure A.1: A comprehensive collection of 21 ARC puzzle input grids used in our analysis, illustrating the diverse visual challenges within the dataset. These examples highlight the variety of tasks the model is designed to address.

A.2 Full Results Table

Task ID	Rate
00576224	0.583
007bbfb7	0.346
009d5c81	0.867
00d62c1b	0.932
00dbd492	0.880
017c7c7b	0.889
025d127b	0.980
03560426	0.930
045e512c	0.834
0520fde7	0.444
05269061	0.388
05a7bcf2	0.587
05f2a901	0.945
0607ce86	0.968
0692e18c	0.827
06df4c85	0.904
070dd51e	0.868
08ed6ac7	0.691
09629e4f	0.818
0962bcdd	0.840
09c534e7	0.849
0a1d4ef5	0.083
0a2355a6	0.906
0a938d79	0.545
0b148d64	0.056
0b17323b	0.978
0bb8deee	0.889
0becf7df	0.850
0c786b71	0.083
0c9aba6e	0.875
0ca9ddb6	0.987
0d3d703e	1.000
0d87d2a6	0.790
0e206a2e	0.980
0e671a1a	0.970
0f63c0b9	0.920
103eff5b	0.829
10fcaaa3	0.850
11852cab	0.950
1190bc91	0.391
1190e5a7	0.600
11dc524f	0.982
11e1fe23	0.952
12422b43	0.814
12997ef3	0.417
12eac192	0.859
13713586	0.518
137eaa0f	0.333
137f0df0	0.920
13f06aa5	0.824
140c817e	0.561
14754a24	0.983
1478ab18	0.578
14b8e18c	0.930
150deff5	0.943
15113be4	0.955
15660dd6	0.516
15663ba9	0.942
15696249	0.395
17829a00	0.781
178fcbfb	0.705
17b80ad2	0.872
17b866bd	0.899
17cae0c1	0.556
18286ef8	0.920
182e5d0f	0.964
18419cfa	0.929
18447a8d	0.920
184a9768	0.628
195ba7dc	0.800
1990f7a8	0.776
19bb5feb	1.000
1a07d186	0.943
1a244afd	0.996
1a2e2828	0.000
1a6449f1	0.750
1acc24af	0.979
1b2d62fb	0.667
1b59e163	0.917
1b60fb0c	0.970
1b8318e3	0.933
1be83260	0.628
1bfc4729	0.830
1c02dbbe	0.409
1c0d0a4b	0.988
1c56ad9f	0.880
1c786137	0.319
1caeab9d	0.900
1cf80156	0.458
1d0a4b61	0.752

Task ID	Rate
1d398264	0.816
1d61978c	1.000
1da012fc	0.905
1e0a9b12	0.560
1e32b0e9	0.592
1e5d6875	0.898
1e81d6f9	0.978
1efba499	0.958
1f0c79e5	0.568
1f642eb9	0.940
1f85a75f	0.188
1f876c06	0.850
1fad071e	0.000
2013d3e2	0.444
2037f2c7	0.650
2072aba6	1.000
20818e16	0.417
20981f0e	0.973
20fb2937	0.657
212895b5	0.919
21f83797	0.840
2204b7a8	0.740
22168020	0.860
22208ba4	0.976
22233c11	0.640
22425bda	0.000
22806e14	0.941
2281f1f4	0.710
228f6490	0.910
22a4bbc2	0.705

Task ID	Rate
22eb0ac0	0.820
230f2e48	0.848
234bbc79	0.792
23581191	0.975
239be575	0.000
23b5c85d	0.000
25094a63	0.894
252143c9	1.000
253bf280	0.936
2546ccf6	0.694
256b0a75	0.462
25c199f5	0.560
25d487eb	0.869
25d8a9c8	0.667
25e02866	0.889
25ff71a9	0.667
2601afb7	0.876
264363fd	0.433
2685904e	0.900
2697da3f	0.713
272f95fa	0.957
2753e76c	0.667
278e5215	0.057
27a28665	0.000
27a77e38	0.951
27f8ce4f	0.383
281123b4	0.188
28bf18c6	1.000
28e73c20	0.531
292dd178	0.412

Task ID	Rate
29623171	0.926
29700607	0.785
29c11459	0.727
2a28add5	0.890
2a5f8217	0.882
2b01abd0	0.812
2b9ef948	0.231
2bcee788	0.940
2bee17df	0.903
2c0b0aff	0.662
2c608aff	0.794
2c737e39	0.836
2ccd9fef	0.164
2dc579da	0.000
2dd70a9a	0.988
2de01db2	0.533
2dee498d	1.000
2e65ae53	0.639
2f0c5170	0.937
2f767503	0.949
2faf500b	0.859
305b1341	0.795
30f42897	0.619
310f3251	0.833
3194b014	0.778
319f2597	0.853
31aa019c	0.910
31adaf00	0.990
31d5ba1a	0.667
320afe60	0.900

Task ID	Rate
321b1fc6	1.000
32597951	0.343
32e9702f	0.320
33067df9	1.000
332202d5	0.984
332efdb3	0.744
3345333e	0.891
337b420f	0.320
3391f8c0	0.844
33b52de3	0.777
3428a4f5	0.467
342ae2ed	0.871
342dd610	0.824
3490cc26	0.945
34b99a2b	0.700
34cfa167	0.796
351d6448	0.410
358ba94e	0.680
3618c87e	0.880
363442ee	0.957
36d67576	0.933
36fdfd69	0.637
37ce87bb	0.748
37d3e8b2	0.951
3906de3d	0.920
396d80d7	0.934
3979b1a8	0.670
39a8645d	1.000
39e1d7f9	0.690
3a301edc	0.601

Task ID	Rate
3aa6fb7a	1.000
3ac3eb23	0.847
3ad05f52	0.597
3af2c5a8	1.000
3b4c2228	0.889
3bd292e8	0.407
3bd67248	0.830
3bdb4ada	0.873
3befdf3e	0.639
3c9b0459	0.222
3cd86f4f	0.812
3d31c5b3	0.722
3d588dc9	0.926
3d6c6e23	0.987
3de23699	0.963
3e980e27	0.846
3eda0437	0.927
3ee1011a	0.327
3f23242b	0.916
3f7978a0	0.550
4093f84a	1.000
40f6cd08	0.718
412b6263	0.261
414297c0	0.348
41ace6b5	0.662
41e4d17e	0.809
423a55dc	0.895
4258a5f9	1.000
4290ef0e	0.603
42918530	0.886

Task ID	Rate
42a15761	0.971
42a50994	0.832
42f14c03	0.643
42f83767	0.406
4347f46a	0.895
4364c1c4	0.918
444801d8	0.810
445eab21	1.000
447fd412	0.784
44d8ac46	0.993
44f52bb0	0.000
4522001f	0.704
456873bc	0.947
45737921	1.000
458e3a53	0.333
45bbe264	0.693
4612dd53	1.000
46442a0e	0.111
465b7d93	0.800
469497ad	0.195
46c35fc7	0.939
46f33fce	0.748
470c91de	0.934
47c1f68c	0.903
48131b3c	0.703
484b58aa	0.801
4852f2fa	0.972
48634b99	0.992
48d8fb45	0.667
48f8583b	0.370

Task ID	Rate
4938f0c2	0.944
494ef9d7	0.312
496994bd	0.967
49d1d64f	0.550
4a1cacc2	0.790
4acc7107	0.750
4b6b68e5	0.710
4be741c5	0.000
4c177718	0.941
4c4377d9	1.000
4c5c2cf0	0.769
4cd1b7b2	0.750
4df5b0ae	0.700
4e45f183	0.474
4e469f39	0.950
4e7e0eb9	0.704
4f537728	0.685
4ff4c9da	0.947
5034a0b5	0.630
505fff84	0.560
506d28a5	0.850
50846271	0.959
508bd3b6	0.854
50a16a69	0.194
50aad11f	0.312
50c07299	1.000
50cb2852	0.973
50f325b5	0.978
5117e062	1.000
5168d44c	0.966

Task ID	Rate
516b51b7	0.906
5207a7b5	0.926
522fdd07	0.895
52364a65	0.859
5289ad53	1.000
52df9849	0.969
52fd389e	0.631
538b439f	0.666
539a4f51	0.690
53b68214	0.830
543a7ed5	0.996
54d82841	0.896
54d9e175	0.649
54db823b	0.978
54dc2872	0.872
55059096	0.955
551d5bf1	0.910
5521c0d9	0.960
5582e5ca	0.889
5587a8d0	0.306
5614dbcf	0.333
5623160b	0.742
56dc2b01	0.778
56ff96f3	0.583
5751f35e	0.812
575b1a71	1.000
5783df64	0.333
5792cb4d	0.990
57aa92db	0.684
57edb29d	0.556

Task ID	Rate
5833af48	0.438
58743b76	0.903
58c02a16	0.188
58e15b12	0.958
59341089	0.417
5a5a2103	0.960
5a719d11	0.685
5ad4f10b	0.222
5ad8a7c0	1.000
5adee1b2	0.744
5af49b42	0.935
5b37cb25	0.539
5b526a93	0.800
5b692c0f	0.847
5b6cbef5	0.777
5bd6f4ac	0.222
5c0a986e	0.940
5c2c9af4	0.684
5d2a5c43	0.833
5d588b4d	0.163
5daaa586	0.383
5e6bbc0b	1.000
5ecac7f7	0.240
5ffb2104	0.950
60a26a3e	0.978
60b61512	0.988
60c09cac	1.000
60d73be6	0.610
6150a2bd	0.444
6165ea8f	0.715

Task ID	Rate
623ea044	1.000
626c0bcc	0.898
62ab2642	0.733
62b74c02	1.000
62c24649	0.889
6350f1f4	0.235
63613498	0.880
639f5a19	0.907
642248e4	0.878
642d658d	0.000
6430c8c4	0.938
6455b5f5	0.760
64a7c07e	0.889
652646ff	0.500
662c240a	0.444
668eec9a	1.000
66ac4c3b	0.901
66e6c45b	0.875
66f2d22f	0.893
67385a82	0.840
673ef223	0.929
67636eac	0.278
6773b310	0.778
67a3c6ac	0.556
67a423a3	0.944
67c52801	0.701
67e8384a	1.000
681b3aeb	1.000
6855a6e4	0.938
689c358e	0.917

Task ID	Rate
68b16354	1.000
68b67ca3	1.000
68bc2e87	0.400
692cd3b6	0.982
694f12f3	1.000
695367ec	0.733
696d4842	0.963
69889d6e	0.980
6a11f6da	0.680
6a1e5592	0.973
6a980be1	0.557
6aa20dc0	0.651
6ad5bdfd	0.820
6b9890af	0.526
6bcdb01e	0.984
6c434453	0.880
6ca952ad	0.847
6cbe9eb8	0.421
6cdd2623	0.984
6cf79266	0.542
6d0160f0	0.967
6d0aefbc	0.667
6d1d5c90	0.250
6d58a25d	0.835
6d75e8bb	0.818
6df30ad6	0.880
6e02f1e3	0.889
6e19193c	0.960
6e82a1ae	0.930
6ea4a07e	0.556

Task ID	Rate
6ecd11f4	1.000
6f473927	0.597
6f8cd79b	0.524
6fa7a44f	0.833
6ffe8f07	0.806
7039b2d7	0.400
705a3229	0.920
712bf12e	0.864
72207abc	0.977
72322fa7	0.898
72a961c9	0.939
72ca375d	0.250
73182012	0.312
73c3b0d8	1.000
73ccf9c2	0.286
7447852a	0.947
7468f01a	0.125
746b3537	0.000
74dd1130	1.000
753ea09b	0.068
758abdf0	0.945
759f3fd3	0.806
75b8110e	0.688
760b3cac	0.926
762cd429	0.500
770cc55f	0.836
776ffc46	0.912
77fdfe62	0.500
780d0b14	0.111
782b5218	0.710

Task ID	Rate
7837ac64	0.667
78e78cff	0.389
79369cc6	0.958
794b24be	1.000
7953d61e	0.266
79cce52d	0.167
7acdf6d3	0.964
7b6016b9	0.869
7b7f7511	0.167
7bb29440	0.500
7c008303	0.750
7c8af763	0.870
7c9b52a0	0.067
7d18a6fb	0.510
7d1f7ee8	0.825
7d419a02	0.838
7d7772cc	1.000
7ddcd7ec	0.800
7df24a62	0.945
7e02026e	0.910
7e0986d6	0.618
7e2bad24	0.898
7e4d4f7c	0.800
7e576d6e	0.741
7ec998c9	0.735
7ee1c6ea	0.830
7f4411dc	0.850
7fe24cdd	0.333
80214e03	0.167
80af3007	0.864

Task ID	Rate
810b9b61	0.993
817e6c09	0.952
81c0276b	0.375
825aa9e9	0.445
82819916	0.786
83302e8f	0.357
833966f4	0.400
833dafe3	0.906
834ec97d	0.799
83b6b474	0.639
83eb0a57	1.000
8403a5d5	0.560
84551f4c	0.819
845d6e51	0.944
846bdb03	0.604
84ba50d3	0.806
84db8fc4	0.940
84f2aca1	1.000
855e0971	0.976
8597cfd7	0.000
85b81ff1	0.962
85c4e7cd	0.020
85fa5666	0.844
8618d23e	1.000
868de0fa	0.833
8719f442	0.947
8731374e	0.275
878187ab	0.727
87ab05b8	0.750
880c1354	0.625

Task ID	Rate
88207623	0.889
8886d717	0.964
88a10436	0.964
88a62173	1.000
890034e9	0.934
891232d6	0.865
896d5239	0.941
8a004b2b	0.714
8a371977	0.821
8a6d367c	0.048
8abad3cf	0.722
8b28cd80	1.000
8ba14f53	0.111
8be77c9e	0.944
8cb8642d	0.781
8d5021e8	0.083
8d510a79	0.890
8dab14c2	0.992
8dae5dfc	0.542
8e1813be	0.551
8e2edd66	0.815
8e301a54	0.990
8e5a5113	0.182
8eb1be9a	0.923
8ee62060	1.000
8efcae92	0.711
8f2ea7aa	0.580
8fbca751	0.688
8fff9e47	0.875
902510d5	0.758

Task ID	Rate
90347967	0.963
90c28cc7	1.000
90f3ed37	0.913
9110e3c5	0.778
913fb3ed	0.965
91413438	0.971
91714a58	1.000
9172f3a0	0.123
917bccba	0.708
928ad970	0.849
92e50de0	0.589
9344f635	0.310
9356391f	0.895
93b4f4b3	0.541
93b581b8	0.389
93c31fbe	0.896
94133066	0.462
941d9a10	1.000
94414823	0.920
9473c6fb	1.000
94be5b80	0.793
94f9d214	0.625
952a094c	0.920
9565186b	1.000
95755ff2	0.719
95990924	0.987
95a58926	0.993
963c33f8	0.969
963e52fc	0.600
963f59bc	0.902

Task ID	Rate
96a8c0cd	0.944
9720b24f	0.929
97239e3d	0.965
973e499e	0.765
9772c176	0.970
97999447	0.917
97a05b5b	0.688
97c75046	0.938
981add89	0.713
9841fdad	0.827
984d8a3e	0.709
985ae207	0.602
98c475bf	0.895
98cf29f8	0.779
992798f6	0.945
99306f82	0.600
995c5fa3	0.000
9968a131	1.000
996ec1f3	0.222
99b1bc43	0.625
99caaf76	0.781
99fa7670	1.000
9a4bb226	1.000
9aec4887	0.816
9af7a82c	0.800
9b2a60aa	0.904
9b30e358	0.640
9b365c51	0.776
9b4c17c4	0.901
9b5080bb	0.750

Task ID	Rate
9ba4a9aa	0.000
9bebae7a	0.909
9c1e755f	0.810
9c56f360	0.988
9caba7c3	0.947
9caf5b84	0.542
9d9215db	0.958
9ddd00f0	0.368
9def23fe	0.932
9dfd6313	0.833
9edfc990	0.906
9f236235	0.562
9f27f097	0.111
9f41bd9c	0.855
9f5f939b	0.915
9f669b64	1.000
9f8de559	0.995
a04b2602	0.979
a096bf4d	0.429
a09f6c25	0.020
a1570a43	0.863
a1aa0c1e	0.467
a2d730bd	0.051
a2fd1cf0	0.917
a3325580	0.583
a3f84088	0.880
a406ac07	0.930
a416b8f3	0.925
a416fc5b	0.868
a48eeaf7	0.970

Task ID	Rate
a5313dff	1.000
a57f2f04	0.794
a59b95c0	0.243
a5f85a15	0.979
a61ba2ce	1.000
a61f2674	0.951
a644e277	0.409
a64e4611	0.638
a65b410d	0.963
a680ac02	0.458
a68b268e	0.938
a6953f00	0.000
a699fb00	0.990
a740d043	0.500
a78176bb	0.880
a79310a0	1.000
a834deea	0.926
a85d4709	0.333
a8610ef7	0.694
a87f7484	0.333
a8c38be5	0.975
a8d7556c	0.938
a934301b	0.840
a9f96cdd	1.000
aa18de87	0.722
aa300dc3	0.990
aa62e3f4	0.600
aab50785	0.469
aabf363d	0.980
aaecdb9a	0.467
aaef0977	0.594
aba27056	0.580
abbfd121	0.972
ac0a08a4	0.986
ac0c2ac3	0.107
ac0c5833	0.941
ac2e8ecf	0.742
ac3e2b04	0.712
ac605cbb	0.876
ac6f9922	1.000
ad173014	0.922
ad38a9d0	1.000
ad3b40cf	0.893
ad7e01d0	0.947
ae3edfdc	0.991
ae4f1146	0.333
ae58858e	1.000
aedd82e4	1.000
aee291af	0.920
af24b4cc	1.000
af726779	0.856
af902bf9	0.970
afe3afe9	0.286
b0722778	0.812
b0c4d837	1.000
b0f4d537	0.333
b15fca0b	0.850
b190f7f5	0.617
b1948b0a	0.875
b1986d4b	0.126
b1fc8b8e	1.000
b20f7c8b	1.000
b230c067	0.930
b25e450b	0.508
b27ca6d3	0.868
b2862040	0.879
b2bc3ffd	0.797
b457fec5	0.655
b4a43f3b	0.941
b527c5c6	0.752
b548a754	0.633
b5bb5719	0.875
b60334d2	0.938
b6afb2da	1.000
b71a7747	0.000
b7249182	0.842
b7256dcd	0.980
b745798f	0.836
b74ca5d1	0.670
b775ac94	0.929
b782dc8a	0.973
b7955b3c	0.439
b7999b51	1.000
b7cb93ac	0.917
b7f8a4d8	0.573
b7fb29bc	0.951
b8825c91	0.875
b8cdaf2b	0.889
b91ae062	0.688
b942fd60	0.758

Task ID	Rate
b94a9452	0.944
b9630600	0.958
b9b7f026	0.000
ba1aa698	0.357
ba26e723	1.000
ba97ae07	0.621
ba9d41b8	0.679
bae5c565	0.983
baf41dbf	0.868
bb43febb	0.980
bb52a14b	0.965
bbb1b8b6	0.438
bbc9ae5d	0.708
bc1d5164	0.778
bc4146bd	0.375
bc93ec48	0.121
bcb3040b	0.926
bd14c3bf	0.898
bd283c4a	1.000
bd4472b8	0.117
bd5af378	0.225
bda2d7a6	0.000
bdad9b1f	0.750
be03b35f	1.000
be94b721	0.083
beb8660c	0.909
bf32578f	0.844
bf699163	1.000
bf89d739	0.903
c074846d	0.949

Task ID	Rate
c0f76784	1.000
c1990cce	0.692
c1d99e64	0.268
c3202e5a	0.920
c35c1b4c	0.850
c3e719e8	0.333
c3fa4749	0.453
c444b776	0.833
c48954c1	0.222
c4d1a9ae	0.838
c59eb873	0.580
c6141b15	0.945
c61be7dc	0.590
c62e2108	0.679
c64f1187	0.901
c658a4bd	0.660
c6e1b8da	0.902
c7d4e6ad	0.970
c803e39c	0.026
c87289bb	0.920
c8b7cc0f	0.778
c8cbb738	0.735
c8f0f002	1.000
c909285e	0.250
c920a713	0.213
c92b942c	0.383
c9680e90	0.909
c97c0139	0.939
c9e6f938	0.778
c9f8e694	0.965

Task ID	Rate
ca8de6ea	0.667
caa06a1f	0.117
cad67732	0.828
cb227835	0.857
cbded52d	0.984
cc9053aa	1.000
ccd554ac	0.299
cce03e0d	0.420
cd3c21df	0.667
cdecee7f	0.333
ce039d91	0.980
ce22a75a	0.765
ce4f8723	0.875
ce602527	0.333
ce8d95cc	0.476
ce9e57f2	0.960
cf133acc	0.924
cf5fd0ad	0.951
cf98881b	0.875
cfb2ce5a	0.630
d017b73f	0.708
d037b0a7	0.556
d06dbe63	0.893
d07ae81c	0.570
d0f5fe59	0.880
d10ecb37	1.000
d13f3404	0.722
d19f7514	0.833
d22278a0	0.671
d23f8c26	0.939

Task ID	Rate
d255d7a7	0.772
d282b262	0.916
d2abd087	0.980
d2acf2cb	0.800
d304284e	0.981
d364b489	1.000
d37a1ef5	0.910
d406998b	0.706
d43fd935	0.890
d4469b4b	0.444
d47aa2ff	0.890
d492a647	0.820
d4a91cb9	0.885
d4b1c2b1	0.009
d4c90558	1.000
d4f3cd78	0.890
d511f180	0.562
d56f2372	0.593
d5c634a2	0.944
d5d6de2d	0.965
d631b094	0.000
d6542281	0.899
d687bc17	0.853

A.3 Detailed Pixel-Level Prediction Analysis

This section presents various visualizations that reveal the model's confidence, the distribution of probabilities and logits for each colour at specific pixels, and a holistic view of the predicted colour likelihoods across the entire output grid. These plots aim to offer transparency into the model's decision-making process, highlighting areas of strong conviction versus uncertainty, and illustrating how it attempts to reconstruct complex patterns.

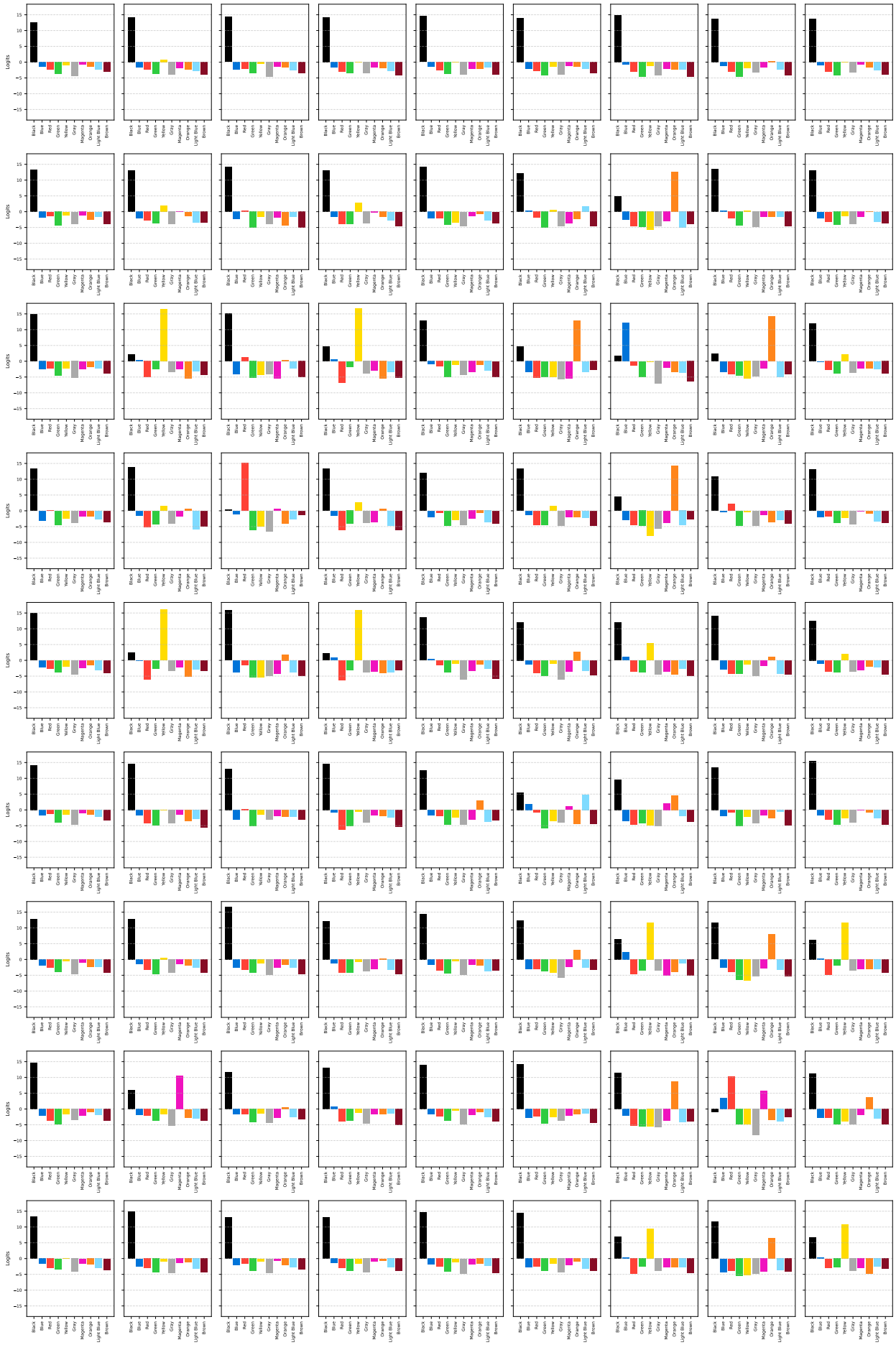


Figure A.2: This show pixel-wise Softmax/Logits for 0ca9ddb6.