Spatially Grounded Compositional Reasoning for Explainable Visual Question Answering

Guoyu Zhang

MInf Project (Part 2) Report
School of Informatics
University of Edinburgh

2025

Abstract

Visual Question Answering (VQA) systems have advanced significantly, but most function as “black boxes" that provide answers without revealing their reasoning processes. Recent research highlights that these systems often exploit statistical biases instead of performing genuine reasoning, lacking faithful grounding in the image. In this work, we propose a framework for spatially grounded compositional reasoning in VQA, explicitly enforcing spatial grounding at every step of the reasoning process. Our method decomposes questions into sequential functional operations using a program generator, while a program executor grounds each operation by predicting bounding boxes around relevant image regions. This mechanism establishes a direct link between reasoning steps and corresponding visual evidence. Experiments conducted on the CLEVR dataset show that our framework achieves a strong correlation between reasoning correctness and answer accuracy, validating faithful reasoning. Furthermore, our approach demonstrates enhanced robustness on the CLEVR-CoGenT dataset, with a notably smaller performance drop (15.3 percentage points) on novel attribute combinations compared to state-of-the-art models (drops of 22.9 and 35.9 points). Detailed analysis reveals that operations requiring minimal spatial reasoning attain high precision, whereas those involving complex spatial reasoning exhibit significantly lower performance.

Acknowledgements

I am grateful to my supervisor Dr. Hakan Bilen for his help and guidance, and friends and family for their support.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Visual Question Answering (VQA) systems have made significant progress in recent years, enabling machines to analyse images and answer natural language questions about their content. However, most current approaches function as “black boxes," providing answers without revealing their reasoning processes or demonstrating how these processes connect to their final outputs.

Research has shown that many VQA systems frequently exploit statistical biases in training data rather than employing genuine reasoning to derive answers (Manjunatha et al., 2019; Geirhos et al., 2020). For instance, when asked "what colour is the sky?", a VQA system might respond "blue" not because it has detected or understood the sky in the image, but simply because the term "sky" is predominantly associated with clear blue skies in the training corpus.

Moreover, existing compositional approaches to VQA, while decomposing questions into sub-tasks, typically lack explicit spatial grounding: their intermediate operations manipulate abstract feature maps or symbolic representations without anchoring each reasoning step back to concrete image regions (Yi et al., 2018; Johnson et al., 2017b; Surís et al., 2023). This disconnect between reasoning steps and visual evidence makes it impossible to verify whether the model is truly examining relevant parts of the image when making decisions.

Such lack of transparency undermines trust in these systems, particularly for applications in critical domains such as healthcare, autonomous driving, and security, where understanding the rationale behind predictions is crucial. As artificial intelligence becomes increasingly integrated into decision-making processes, the need for explainable, faithful, and trustworthy visual reasoning systems has never been more essential.

1.2 Contribution

We introduce a novel framework for spatially grounded compositional reasoning in visual question answering that addresses these limitations. Our primary contribution is enforcing spatial grounding at each reasoning step by requiring our model to predict bounding boxes around relevant objects as it executes a program.

Like other compositional VQA approaches, our framework decomposes questions into sequential functional steps, but uniquely grounds each step to specific image regions, creating a direct link between reasoning operations and visual evidence. This mechanism ensures that the model genuinely refers to the image content when generating responses, rather than exploiting statistical patterns or biases in the training data. Our approach provides faithfulness and transparency by visually demonstrating which parts of the image influence each reasoning operation.

In experiments on the CLEVR dataset, while our framework does not achieve state-of-the-art (SOTA) accuracy, it demonstrates an increased performance over the baseline. Notably, our approach shows more robust generalisation to novel combinations of visual attributes, with less performance degradation when faced with unfamiliar attribute combinations compared to SOTA models. Additionally, our analysis reveals a strong correlation between reasoning correctness and answer accuracy: when our model's reasoning process fails, the answer typically fails as well, confirming that our framework's answers genuinely depend on the reasoning process and the image rather than relying on shortcuts or spurious correlations.

1.3 Structure

The remainder of this report is organised as follows. Chapter 2. reviews deep learning methods relevant to our approach, VQA methods, explainable AI techniques, and related work. Chapter 3 details our methodology, including dataset preprocessing, model architectural, loss functions used, and evaluation metrics. Chapter 4 presents our experiments: we evaluate on interpretability and accuracy, data efficiency and generalisation abilities. Chapter 5 summarises our findings, discusses limitations and avenues for future research.

Chapter 2

Background

This section provides the necessary background for understanding the proposed approach for spatially grounded compositional reasoning in explainable visual question answering. We will first review relevant deep learning methods, followed by an overview of visual question answering, explainable artificial intelligence, and finally, a discussion of related work in the field.

2.1 Deep Learning Methods

This work utilises various deep learning methods to build a visual question answering framework. We provide this background to familiarise readers with modern deep learning approaches relevant to our methods, while offering a concise overview for those new to the field.

Deep learning is a subset of machine learning that focuses on learning representations from data using artificial neural networks with many layers, hence the term “deep." Deep learning has revolutionised applications across diverse industries and research domains, including healthcare (Shamshirband et al., 2021), agriculture (Kamilaris and Prenafeta-Boldú, 2018), and robotics (Pierson and Gashler, 2017). Several architectural innovations have enabled these advances, particularly Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformer architectures, which we briefly review in the following sections.

2.1.1 Convolutional Neural Networks

CNNs are a class of deep learning models specifically designed for grid-structured data like images (LeCun et al., 1998). A CNN uses convolutional layers with learned filters that act as local feature detectors, scanning across the input image to produce feature maps. By exploiting local spatial correlations and sharing weights across spatial locations, CNNs capture simple patterns, such as edges or textures, in early layers and more complex structures in deeper layers. Pooling layers are often inserted to progressively reduce spatial resolution and achieve a degree of translation invariance (LeCun et al., 1998).

CNN architectures have been tremendously successful in visual tasks. A seminal example is AlexNet, which achieved a breakthrough on the ImageNet challenge by using a deep CNN to dramatically improve image classification accuracy (Krizhevsky et al., 2012). This result sparked the development of even deeper CNNs, such as the 19-layer VGG network (Simonyan and Zisserman, 2014) and the ResNet architecture which introduced residual connections to enable training of networks over 100 layers with superior performance (He et al., 2016). CNNs have consequently become the backbone of most computer vision systems, including VQA pipelines where a pretrained CNN is used to extract visual feature representations from images (Antol et al., 2015).

2.1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are designed to model sequential data by maintaining a hidden state that is updated at each time step of a sequence (Elman, 1990). This recurrence allows information to persist across timesteps, making RNNs well-suited for tasks where tokens depend on earlier tokens, like language modelling or time series analysis. However, training traditional RNNs on long sequences is challenging due to vanishing and exploding gradient problems, which make it difficult for the network to learn long-term dependencies (Bengio et al., 1994).

More advanced RNN architectures introduce gating mechanisms to address these limitations. The Long Short-Term Memory (LSTM) network (Hochreiter and Schmidhuber, 1997) augments the recurrent cell with 3 gates and an internal memory cell that together regulate the flow of information, enabling the network to learn dependencies over much longer time frames. Another extension, the bidirectional RNN, processes the sequence in both forward and reverse directions and then combines the two pass representations (Schuster and Paliwal, 1997). Bidirectional LSTMs have been particularly successful in tasks where context from both past and future tokens is beneficial.

Despite their strength in sequential modelling, RNNs have notable disadvantages. Because the hidden state is propagated one timestep at a time, RNN computation is inherently sequential, which limits the ability to parallelise across sequence positions (Vaswani et al., 2017). This sequential bottleneck leads to slower training and makes it difficult to capture very long-range dependencies, these challenges set the stage for the development of alternative sequence modelling architectures, most notably the Transformer.

2.1.3 Transformers

Proposed by Vaswani et al. (2017) initially for machine translation, Transformers have emerged as a powerful architecture for sequence modelling. The core innovation of the Transformer is it's self-attention mechanism. In self-attention, each element of the input attends to every other element, allowing the model to weigh their importance when building representations.

Self-attention is permutation-invariant (it does not inherently encode sequence order), thus Transformers introduce positional encodings added to the input embeddings to

provide the model with information about the order of tokens. These positional encodings can be fixed or learned vectors, and they ensure that the model can distinguish the positions of otherwise identical tokens in a sequence.

The Transformer is composed of stacked layers that each contain a multi-head self-attention sublayer and a position-wise feed-forward sublayer, with residual connections and normalisation applied. This design can be used in an encoder–decoder structure (for tasks like translation) or just an encoder or decoder alone for tasks like classification or generation.

Transformers have achieved remarkable results across modalities. In natural language processing, large pre-trained Transformer models such as BERT (Devlin et al., 2019) and the Generative Pre-trained Transformer (GPT) series (Brown et al., 2020) have set new benchmarks natural language understanding tasks. In computer vision, the Vision Transformer (ViT) demonstrated that a Transformer applied to image patches can attain competitive image classification performance, matching or surpassing convolutional networks when trained on sufficient data (Dosovitskiy et al., 2021). Furthermore, Transformer-based architectures have been applied to structured prediction tasks in vision: for example, the DETR model uses a Transformer encoder–decoder to perform object detection in an end-to-end fashion (Carion et al., 2020).

2.2 Visual Question Answering

Visual Question Answering (VQA) is a multidisciplinary task that involves computer vision, natural language processing, and knowledge representation and reasoning. The objective of VQA is to interpret an image and generate accurate natural language answers to questions asked about that image. This task is complex, as it encompasses a range of sub-tasks including object recognition, identifying spatial relationships between objects, counting objects, and applying commonsense reasoning (Kafle and Kanan, 2017).

VQA questions come in diverse forms, such as multiple choice (Zhu et al., 2016), binary (yes/no or true/false) (Antol et al., 2015), or open-ended questions (Xu et al., 2019). In addition to varied question types, another challenge for VQA is dataset bias. This is where certain questions are predictable without relying heavily on image content. For instance, given the questions “what colour is the banana", the model might be heavily biased towards “yellow" without thoroughly taking the image into account, just because this relationship was frequently observed in the training data. Furthermore, high quality datasets are essential for VQA but are challenging and resource-intensive to produce. As a result, many VQA datasets are either low-resolution, limited in scope, or simplified to facilitate human annotation, which can hinder model training and real-world applicability (Sharma and Jalal, 2021).

2.2.1 Compositional Approaches

As mentioned, the inherent complexity of Visual Question Answering (VQA) tasks often necessitates multiple steps of reasoning. For example, if a question asks “What

is on the left of the green ball?" the model must first identify the green ball and then determine the object located to its left. By breaking down such questions into a sequence of sub-tasks (e.g., object detection, spatial reasoning), each step can be addressed by specialised neural sub-networks. These independent modules are then combined in a well structured sequence to answer the question.

Andreas et al. (2016) first introduced this approach for VQA. Their approach uses a natural language parser to decompose each question into sub-tasks and dynamically build the network needed to answer it. They use the Stanford dependency parser (De Marneffe and Manning, 2008) to extract relationships between words, allowing the model to assemble a program layout, which is a sequence of neural modules corresponding to each sub-task. Five modules are used to represent core tasks and are implemented as shallow neural networks. Each module is designed to either produce an attention map or output a final answer. This modular design not only enables the model to provide reusable and compositional behaviour but also handle complex VQA tasks with better interpretability.

Later works such as Compositional Attention Network (Hudson and Manning, 2018) improves this by removing the need to individual neural models. This model breaks down the reasoning process into a series of attention-based reasoning steps through the use of repeated memory attention and composition (MAC) cells. Each MAC cell maintains control and memory states, allowing the model to update an intermediate result based on attention of certain words and phrases at each time step.

2.2.2 Visual Grounding

Visual grounding is the process of connecting linguistic expressions with specific regions or objects in an image. For example, given a language expression like “red ball," a visual grounding system should identify the corresponding bounding box of the red ball in the image. Early approaches to visual grounding built on object detection pipelines, with Faster R-CNN (Ren et al., 2016) being a prominent method. Faster R-CNN can propose dozens of likely object regions per image, which are then classified into object categories. These region proposals have been widely used in vision-and-language tasks, for instance: VQA models often incorporate features from the top N detected object regions as a "bottom-up" attention mechanism. This strategy, first introduced by Anderson et al. (2018), enables models to focus on salient objects when answering questions, significantly improving VQA performance by providing a set of discrete, content-rich image regions for the model to attend to.

In recent years, visual grounding has been improved with transformer architectures and multi-modal learning. Instead of treating detection and grounding as separate stages, newer models unify these processes. For example, MDETR (Modulated Detection Transformer) by Kamath et al. (2021) reformulates object detection as a multi-modal task: given an image and a text query, MDETR uses a transformer encoder-decoder (based on DETR (Carion et al., 2020)) to directly output bounding boxes for objects mentioned in the text. This end-to-end approach integrates visual and linguistic features, the text modulates the detection process so that only referred objects are detected. MDETR was trained on combined datasets with region-text annotations and achieved

state-of-the-art results on phrase grounding and referring expressions. These grounded vision-language models provide explicit alignments between words and image regions, which are extremely useful for tasks like VQA that require identifying what in the image a question refers to and where that information is located in the image.

2.3 Explainable Artificial Intelligence

The rise of deep learning has led to significant advancements in the integration of machine learning (ML) models to industry applications, enhancing task efficiency and productivity. These models have demonstrated remarkable capabilities in domains from speech processing, time series forecasting to medical diagnosis (Huang et al., 2024; Bhangale and Mohanaprasad, 2021; Kononenko, 2001). However, their complex architectures pose challenges for interpretability, and thus they are commonly termed black box models. It is difficult to determine whether these models are actually identifying relevant patterns in the data to solve the intended problem or merely exploiting incidental correlations (Manjunatha et al., 2019; Geirhos et al., 2020). This lack of transparency in decision making could undermine trust, especially in critical applications within healthcare, government, defense, and finance, where understanding the rationale behind predictions is crucial.

The demand for interpretable models has led to the emerging field of explainable artificial intelligence (XAI), which aims to provide insights into model behaviour and ensure alignment with human preferences. In certain domains, it is legally required to explain the decision making process of these models (Regulation, 2016). Beyond this, explainability can also enhance model performance, improve efficiency, and uncover new data-driven insights (Yeom et al., 2021; Schütt et al., 2017). Current XAI techniques range from perturbation-based methods, gradient-based methods, to explanation-based methods.

2.3.1 Methods in Explainability

2.3.1.1 Perturbation-Based Methods

Perturbation-based approaches explain a model by tweaking inputs and observing how its output changes. LIME (Ribeiro et al., 2016) builds a sparse linear surrogate around the perturbed samples; its coefficients reveal each feature's influence on the specific prediction. SHAP (Lundberg, 2017) uses Shapley values to apportion credit across all feature coalitions, offering local and global importance at greater computational cost. In NLP, for example, masking words shows their contribution to sentiment decisions.

2.3.1.2 Gradient-Based Methods

Gradient-based techniques derive saliency from the gradient of the output with respect to the input. Saliency maps (Simonyan, 2013) highlight pixels with the largest gradient magnitudes, outlining influential regions. Class Activation Mapping (CAM) (Zhou et al., 2016) weights the final convolutional feature maps by class-specific gradients, producing a coarse heat-map that localises class-relevant areas.

2.3.1.3 Textual Explanation Methods

Some models generate natural-language rationales alongside predictions. TED (Hind et al., 2019) reframes explainability as a supervised learning task by asking experts to annotate each training example not only with its label but also with a concise, human-readable rationale. At training time, these label–explanation pairs are combined into a single target class; at inference, the model predicts both the decision and its accompanying explanation. Because the explanations come directly from domain experts, they naturally match the audience's vocabulary and complexity, and—crucially—adding them often boosts prediction accuracy rather than hindering it.

2.3.2 Explainability in Visual Question Answering

VQA models take an image and a question and produce an answer – but how they arrived at that answer is often a "black box." Explainability in VQA is a growing area, aiming to make the reasoning process transparent and the model's answers more trustworthy Li et al. (2025). Many recent approaches have been proposed to provide explanations for VQA outputs, ranging from visual attention maps to generated textual justifications. Below, we discuss recent explainability techniques.

A simple way to explain a VQA prediction is to highlight the image regions the model "looked at" when answering the question. Most VQA models use attention mechanisms to fuse visual and textual information, yielding attention weights over image regions. These attention maps can be overlaid as heatmaps on the image as a form of explanation (Li et al., 2025). Recent analyses have shown that attention weights are not necessarily faithful indicators of importance. Liu et al. (2022) introduced a faithfulness violation test for attention-based explanations and discovered that in some VQA models, removing the top attended regions can increase confidence in the answer. This issue has motivated the development of more faithful visual explanation methods. Some techniques combine attention with gradient-based attribution, such as Attention*x*Gradient or guided backpropagation on the attention layers, to identify which visual features truly impacted the answer (Liu et al., 2022).

Another line of work produces textual explanations. Datasets like VQA-X (Park et al., 2018) provide human-written explanations for some QA pairs, allowing models to be trained to generate similar justifications. Vaideeswaran et al. (2022) augments a VQA model with an explanation generation module, using either an LSTM or a transformer decoder to produce a sentence rationale. However, a challenge is that models can generate fluent sentences that sound reasonable but are not truly reflective of the model's reasoning. Thus, ensuring faithfulness in textual explanations is crucial. Some approaches enforce that the explanation must mention the visual attributes the model actually used, and others use training strategies where the model is penalised if the explanation does not correlate with known important features (Lai et al., 2023).

Furthermore, there also exists symbolic reasoning approaches applied to explainability. For instance, some VQA systems convert the question into a symbolic program and execute it on a structured representation of the image (Yi et al., 2018). While the primary goal of such systems is accurate reasoning, a side benefit is that the program

itself serves as an explanation. A user can inspect this program to understand the logic applied.

2.4 Related Work

To build VQA systems that are both interpretable and capable of complex reasoning, researchers have explored methods that explicitly incorporate compositional structures. Here we compare several approaches, including neural module networks, neuro-symbolic reasoning, and recent "visual programming" techniques.

2.4.1 Inferring and Executing

Johnson et al. (2017b) introduced IEP (Inferring and Executing Programs for Visual Reasoning) which builds on neural module networks. Their system has two neural components: a program generator that translates a natural language question into a sequence of functional tokens and an execution engine that maps each token to a generic CNN-based module and combines the modules into an end-to-end differentiable network which is then executed to obtain the answer.

IEP offers a high level of interpretability as the predicted program is an explicit, human-readable reasoning trace and each module's intermediate attention can be visualised. However, a drawback of this approach is that it can be difficult to ascertain whether each module is actually performing as intended, since the execution engine applies them directly to the image and then sequentially executes the predicted reasoning steps to derive the answer, making it hard to verify the correctness of each stage.

While IEP exposes a readable program trace, the intermediate feature maps it passes between modules are not forced to align with tangible image evidence. Our executor instead outputs verifiable bounding boxes at every spatial step, so the reasoning chain is faithful—each operation is anchored to pixels the model actually used.

2.4.2 NS-VQA

NS-VQA (Neural-Symbolic VQA) (Yi et al., 2018) attempts to improve on IEP. They first uses neural networks to convert the image into a structural representation, such as a scene graph, and the question into a functional program. Then custom Python modules execute the program sequence using the structural representation. In terms of interpretability, one can examine the intermediate structured representation and each step of the program execution.

The downside is that NS-VQA relies on having high quality symbolic representations. If the vision system misses an object or mislabels something, the symbolic reasoner will fail on questions about it. Furthermore, as the framework depends on the vision system to construct representations, the vision system may be prone to learning biases in data. Both these issues lead to problems in generalisation.

NS-VQA's symbolic pipeline can drift from the image whenever the pre-computed scene graph is wrong. By re-localising objects and propagating their boxes through the

chain, our system maintains faithfulness so that every reasoning error is traceable to a specific mis-grounded region rather than an opaque graph mismatch.

2.4.3 Visual Programming

Visual Programming is a class of methods that leverages large language models (LLMs) to generate executable programs for visual tasks. An example is ViperGPT (Visual Inference via Python Execution for Reasoning) (Surís et al., 2023). This approach uses an LLM to write a short Python program calling a set of pre-defined vision functions, given a question about an image. The interpretability of this approach is high: the program itself can be considered a step-by-step explanation of how the answer was obtained. If the answer is wrong, one can debug by inspecting the program or the outputs of each function call.

In terms of performance, visual programming can leverage very powerful pre-trained models as tools which means the system's capabilities are tied to those of its tools. In zero-shot settings, ViperGPT demonstrates the ability to handle complex multi-step questions that stump typical end-to-end models. The generalisation here is largely due to the LLM's ability to understand the question and logically compose tools. This approach also allows for additional explanations to be generated, for example, the system could be asked to output the intermediate results or a textual summary of the program.

Visual programming methods rely on external tools whose internal heuristics may not reflect what is visible in the current frame. Oue generator–executor pair ties each program step to an explicit bounding box output, guaranteeing that the explanation remains causally faithful to the image.

Chapter 3

Methodology

This section outlines the core components of our framework. First, we describe the dataset and the preprocessing pipeline. Next, we present the overall architecture, detailing both the **program generator** and the **program executor**, alongside the loss functions used to optimise each. We then introduce a baseline model architecture for comparison. Finally, we summarise our evaluation metrics and specify the compute environment used for all experiments.

3.1 Approach Overview

Many machine learning models operate as black boxes, providing answers without revealing their reasoning processes or demonstrating how their reasoning connects to their final outputs. Research has shown that visual question answering (VQA) systems frequently exploit statistical biases in training data rather than employing genuine reasoning to derive answers (Agrawal et al., 2016). For instance, when asked “what colour is the sky?", a VQA system might respond “blue" not because it has detected or understood the sky in the image, but simply because the term "sky" is predominantly associated with clear blue skies in the training corpus.

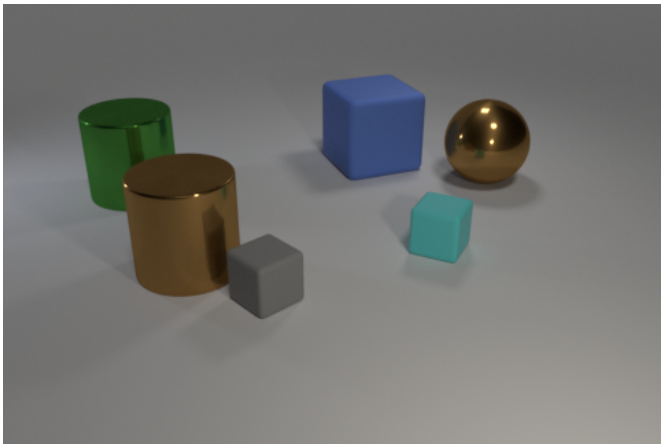
Our proposed approach addresses these limitations through two key strategies: decomposing the VQA task into multiple sequential steps and grounding the model’s reasoning to the image. To accomplish this decomposition, we implement a framework consisting of two components: a program generator and a program executor. The program generator produces program steps when presented with an image and a question, while the program executor processes these steps sequentially to arrive at the final answer. For example, if the question asks to identify how many red objects are in an image, the task might be decomposed into first locating all red objects and then counting them.

To ground the model’s reasoning to the image, we incorporate bounding boxes into our program executor. During reasoning steps that require specific visual examination (such as identifying red objects), the model must predict bounding boxes around relevant objects. This mechanism ensures that the model is genuinely analysing the image content to generate responses rather than exploiting patterns or biases in the training

data. Naturally, training such a framework requires data that includes images, questions, program steps, and the input and output at each individual step.

3.2 Data Description

This project utilises the CLEVR dataset (Johnson et al., 2017a), which was designed to evaluate visual reasoning capabilities in visual question answering (VQA) systems. The dataset comprises 100,000 synthetic images containing simple geometric objects, with approximately 10 corresponding questions per image, totaling around 1,000,000 questions. The data is partitioned into three sets: a training set (70,000 images, 699,989 questions), a validation set (15,000 images, 149,991 questions), and a test set (15,000 images, 14,988 questions). Figure 3.1 illustrates a sample image with a few of its corresponding questions.

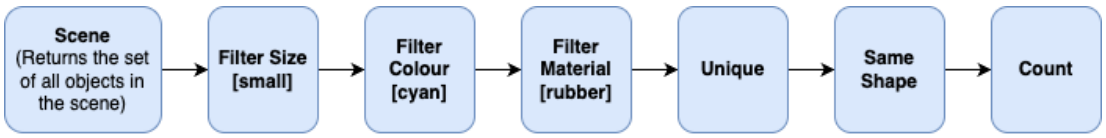


Question 1: Are there more big green things than large purple shiny cubes?
Question 2: How many other things are there of the same shape as the tiny cyan matte object?
Question 3: Is the colour of the large sphere the same as the large matte cube?
Question 4: Are there any other things that have the same size as the brown shiny sphere?

Figure 3.1: A sample image and 4 corresponding questions from CLEVR.

Each image in the dataset features a varying number of objects. These objects come in three shapes (cube, cylinder or sphere), eight colours (blue, green, cyan, red, yellow, purple, gray and brown), two materials (metal and rubber), and two sizes (small and large). Every image also has a scene graph that records all the objects' attributes and where they're located.

The questions in the dataset are constructed with functional programs that can be executed on an image's scene graph to produce answers. The programs are built from composing basic functions that perform simple visual reasoning operations like querying object attributes, counting objects, or comparing values. Examples of a few functions are as follows: query_attribute (returns a specified attribute of an object), count (returns the number of elements in a set), and compare (performs numerical or attribute comparison between inputs). A simple example of a question and the associated program is shown in Figure 3.2.



Question 2: How many other things are there of the same shape as the tiny cyan matte object?

Figure 3.2: An example of a question and the associated program, where the program executes the following steps: Scene (returns all objects in the scene), Filter Size [small] (filters for small objects), Filter Colour [cyan] (further filters for cyan objects), Filter Material [rubber] (filters for rubber objects), Unique (ensures we have a single unique object), Same Shape (finds all objects with the same shape as the reference object), Count (returns the number of objects in the result set).

Each question entry in the dataset contains numerous metadata fields, such as question_index, image_index, and question_family_index, among others. However , the core fields relevant to our work are: **question** (the natural language question about the image), **image_filename** (reference to the corresponding image file), **answer** (the ground truth answer), and **program** (the sequence of functional operations). Table 3.1 shows an example of these fields.

Field	**Value**
question	Are there more big green things than large purple shiny cubes?
answer	yes
image_filename	CLEVR_train_000000.png
program	[{inputs: [], function: "scene", value_inputs: []}, {inputs: [0], function: "filter_size", value_inputs: ["large"]}, ..., {inputs: [3,9], function: "greater_than", value_inputs: []}]

Table 3.1: Example of core data fields for a CLEVR question

The functional programs in CLEVR are particularly valuable for our work as they expose the underlying reasoning process required to answer each question, which can be directly incorporated into our program generator and executor framework. As shown in Table 3.1, each program consists of a sequence of functional steps. Each step includes a function name (e.g., *scene*, *filter_size*, *filter_material*), an input index referencing previous steps whose outputs serve as inputs for the current step, and value_inputs which provide additional parameters for the function (e.g., "large", "metal").

Lastly, in addition to the main CLEVR dataset, Johnson et al. (2017a) also introduced a Compositional Generalisation variant (CLEVR-CoGenT) to evaluate models' ability to recognise novel combinations of attributes at test time. This dataset is separated into two splits: Split A contains cubes that are either gray, blue, brown, or yellow, and cylinders that are red, green, purple, or cyan; Split B has the opposite colour-shape pairs. Split A includes 70,000 images and 699,960 questions for training, 15,000 images and

150,000 questions for validation, and 15,000 images and 149,980 questions for testing. Split B contains 15,000 images and 149,991 questions for validation and 15,000 images and 14,992 questions for testing.

It should be noted that for both the main CLEVR and CLEVR-CoGen datasets, the official test sets do not include answer annotations, as the authors required researchers to submit model predictions for evaluation. Since this project is no longer actively maintained, we instead created our own validation and test split from the provided validation set. This process is described in Section 3.3 (Data Preprocessing).

3.3 Data Preprocessing

To prepare the CLEVR dataset for our program generator and executor framework, several preprocessing steps were necessary. The primary challenge was transforming the original functional programs into a format suitable for sequential execution with explicit inputs and outputs at each step.

3.3.1 Function Parameterisation

To simplify the model design, we adopted a strategy of treating parameterised functions as distinct function types. For example, rather than having a single function `filter_size` that accepts parameters such as `large` or `small`, we separated these into individual functions: `filter_size[large]` and `filter_size[small]`. This approach eliminates the need for the model to take in both the function and its parameters separately, effectively reducing the complexity of the prediction task whilst maintaining the full expressivity of the original functional program structure.

3.3.2 Program Execution Processing

The original dataset provides programs as sequences of functional steps, where each step references previous steps' outputs as inputs through indices. To enable independent training of each program step, we implemented Python functions that replicate the logic of each function described in the supplementary material of CLEVR (Johnson et al., 2017a).

Since later steps in a program depend on outputs from earlier steps, we processed each program sequentially for both the CLEVR and CLEVR-CoGenT datasets. This involved executing steps in order, storing intermediate outputs, and passing these stored outputs as inputs to subsequent steps when required by the program structure.

To facilitate the prediction of program steps and their corresponding input indices, we created a new field that contains the entire program chain. To separate each step within the chain, we introduced separator tokens (`<SEP>`), creating a structured format suitable for training our program generator.

3.3.3 Bounding Box Approximation

Different functions in the program sequence produce different types of theoretical outputs. Some functions, such as *query_colour*, output specific attribute tokens (e.g., "red"), while others like *scene* output spatial information about objects in the form of bounding boxes

A significant challenge is that the CLEVR dataset does not contain ground truth bounding boxes. Instead, it provides scene graphs with attributes such as 3D coordinates, rotation, and pixel coordinates. To address this limitation, we developed an approximation method to derive bounding boxes for all objects in the scenes. The detailed process for this approximation is described in Appendix A.1.

3.3.4 Image Preprocessing

For the image component of our framework, we employed a ResNet-101 (He et al., 2016) backbone pretrained on ImageNet (Deng et al., 2009). To ensure compatibility with this architecture, all images were resized to 224×224 pixels and normalised using the standard ImageNet normalisation parameters with mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.224]. When the images are resized to 224×224 pixels, we scale our approximated bounding boxes accordingly so they maintain the same spatial positions.

3.3.5 Dataset Splitting

As mentioned in Section 3.2, the original test sets for both CLEVR and CLEVR-CoGenT do not include answer annotations, and we are unable to obtain annotations as the project is no longer actively maintained. To address this limitation, we created our own validation and test splits from the provided validation set.

For the CLEVR dataset, we extracted 14,988 questions (matching the size of the original test set) from the validation set to create our test set, ensuring that the same proportion of question types was maintained for meaningful evaluation. This left 135,003 questions for our validation set. Similarly, for CLEVR-CoGenT, we extracted 14,992 questions to form our test set, leaving 134,999 questions in the validation set. For both datasets, we maintained the original image count of 15,000 for each split.

3.4 System Architecture

Our approach utilises two models, a program generator, to produce individual program steps, and a program executor, which will run these steps sequentially to produce an answer. During training, the program generator and program executor are separately trained end-to-end, but during inference we implement a cache setup that stores the output values of each step for efficient lookup. This allows subsequent steps to access previous outputs as inputs without redundant computation. An illustration of the overall architecture is shown in Figure 3.3.

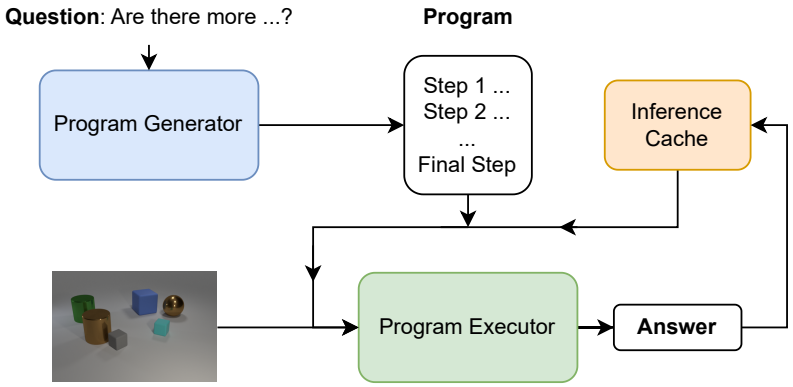


Figure 3.3: Simple overview of our framework. The program is fed in to the program executor step by step. During inference, intermediate inputs and outputs are saved and retrieved from the inference cache.

3.4.1 Program Generator

The program generator takes in a question to predict the program sequence using an encoder-decoder architecture with attention. The encoder uses a bidirectional Long Short-Term Memory (LSTM) architecture to produce contextualised word representations. The decoder then uses these representations to generate the program sequence step by step, using a dot-product attention mechanism similar to Luong et al. (2015). The model is trained using cross-entropy loss and features 3 encoder layers and 3 decoder layers, with hidden dimensions and word embedding dimensions set to 512 and 300, respectively. This architecture allows the model to understand the question and convert it into a structured program by maintaining important information between steps.

3.4.2 Program Executor

The program executor employs a transformer architecture that processes three inputs: the image, the function, and any dependencies from previous steps (which may be either bounding boxes, two tokens, or empty), outputting predictions in the form of either bounding boxes or a single token, as illustrated in Figure 3.4.

Inference Cache During inference, we implement a cache system for function function outputs and image features, stored in a hashmap to enable fast retrieval. This cache maintains output values for each program step, as subsequent steps may utilise output values from earlier steps as input. Additionally, the image features are cached to avoid redundant processing through the image encoder at each step, allowing direct fusion with the bounding boxes and tokens/functions.

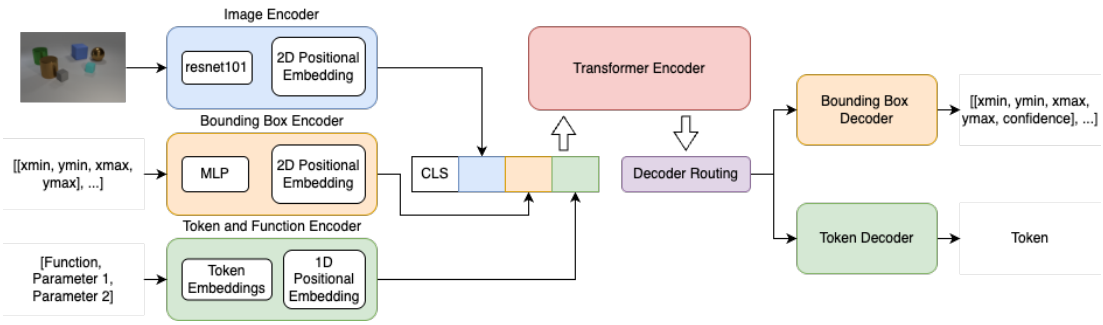


Figure 3.4: Model architecture of the program executor.

3.4.2.1 Image Encoder

To extract image features, we employ a pretrained ResNet101 architecture (He et al., 2016) that was previously trained on ImageNet (Deng et al., 2009). We feed the input image through this encoder and extract the feature map produced at the conv4 block. Specifically, for an input image with resolution $H \times W$ and $C = 3$ colour channels, the conv4 output generates a feature map with shape:

$$F \in \mathbb{R}^{C \times H' \times W'}, \quad C = 1024,\ H' = \frac{H}{16},\ W' = \frac{W}{16}$$

where F represents the feature map, C is the number of channels, and H' and W' denote the reduced spatial dimensions of the feature map. We then apply a 1×1 convolution to project the channel dimension down to match our transformer encoder's hidden dimension d, resulting in a feature map of shape (d, H', W'), where d is the transformer hidden dimension.

Since transformer architectures are permutation invariant, we incorporate positional information by adding fixed 2D sine-cosine positional embeddings to the image features, following the approach in Kamath et al. (2021). For a feature map of shape (d, H', W'), we construct a positional encoding tensor of identical dimensions and add it element-wise to the image features. This is accomplished by allocating half the d channels for encoding the x-axis position and half for the y-axis position. For each spatial location (i, j) and each channel index $k \in \{0, \ldots, \frac{d}{2} - 1\}$, we calculate two signals:

$$\begin{aligned} \mathrm{PE}_x(2k, i, j) &= \sin\left(\frac{i}{10000^{2k/d}}\right), & \mathrm{PE}_x(2k+1, i, j) &= \cos\left(\frac{i}{10000^{2k/d}}\right), \\ \mathrm{PE}_y(2k, i, j) &= \sin\left(\frac{j}{10000^{2k/d}}\right), & \mathrm{PE}_y(2k+1, i, j) &= \cos\left(\frac{j}{10000^{2k/d}}\right). \end{aligned} \tag{3.1}$$

We then concatenate PE_x and PE_y along the channel dimension to create a complete (d, H', W') positional tensor. By adding this tensor element-wise to the projected image features, the transformer acquires both absolute location information—through distinct sine/cosine patterns at different (i, j) coordinates—and relative distance information, as differences in these encodings reflect spatial relationships between positions.

Finally, we flatten the spatially-encoded image features to obtain a tensor of shape $(d, H'W')$, which will later be combined with the bounding box embeddings and token and function embeddings in the transformer encoder.

3.4.2.2 Bounding Box Encoder

To encode bounding boxes, we implement a simple multi-layer perceptron (MLP) architecture. This encoder processes bounding boxes consisting of four coordinates as described in Section 3.3.3, and produces embeddings that match the hidden dimension of the transformer encoder used in later stages.

The MLP has two layers with hidden dimension d (same d as our transformer dimensions), and employs ReLU activation functions for non-linearity. After encoding the box coordinates, we add 2D fixed positional embeddings following the same approach used for image features in Section 3.4.2.1, with one important adaptation. While in Section 3.4.2.1 we had spatial locations (i, j) for all positions in the $H' \times W'$ grid, here we use the normalised centre coordinates of each bounding box, which range between 0 and 1.

To encode the spatial position of each box in the same coordinate frame as our $H' \times W'$ image feature grid, we convert the continuous centre (x_c, y_c) into feature-map coordinates:

$$i_c = \frac{y_c}{H}(H'-1), \quad j_c = \frac{x_c}{W}(W'-1), \quad i_c, j_c \in [0, H'-1] \times [0, W'-1].$$

We then apply the same sine-cosine formulas as in Equation 3.1 to generate PE_x and PE_y (each of dimension $d/2$), which we concatenate along the channel dimension to create the full positional tensor. This tensor is added element-wise to the MLP output to produce the final box token.

Since inputs may contain up to 10 bounding boxes or potentially none, we process the bounding boxes in batches and pad the sequence to a fixed length of 10 using a learned [PAD] embedding. We then mask out the padded positions in the Transformer encoder. If no bounding boxes are provided, all 10 slots are set to `[PAD]` and masked out, allowing the transformer encoder to effectively ignore the box-embedding stream and process only the image features and token/function encodings.

3.4.2.3 Token and Function Encoder

For the programs in our data, we process input tokens representing function steps in the program, which are sometimes accompanied by two additional argument tokens. This results in a sequence of three tokens: $\left[t^{(\text{func})}, t^{(1)}, t^{(2)}\right]$, where $t^{(\text{func})}$ is the function token, and $t^{(1)}$ and $t^{(2)}$ are the argument tokens. In cases where arguments are not required, we fill these positions with [PAD] tokens.

We learn an embedding matrix that maps these tokens to a d-dimensional embedding space, where d is the hidden dimension of the transformer encoder. Additionally, we incorporate learned positional embeddings, which are crucial for enabling the transformer to distinguish between the function token and its arguments, and to recognise the order of arguments—particularly important for functions that compare them. These positional embeddings are also learned and have a dimension of d.

For each position in our sequence of three, we compute the element-wise sum of the token embedding and the positional embedding, as shown in Equation (3.2):

$$\mathbf{e}_j = \text{TokenEmbed}(t_j) + \text{PosEmbed}(j) \quad \in \mathbb{R}^d, \quad j = 1,2,3. \tag{3.2}$$

We stack these three embeddings into a tensor $\mathbb{R}^{3\times d}$ and construct a boolean mask $m \in \{0,1\}^3$, where $m_j = 0$ for any `[PAD]` position, ensuring that the transformer encoder ignores these positions. The embedded tokens and function are subsequently combined with the image features and bounding box features in the transformer architecture.

3.4.2.4 Multimodal Fusion Transformer Encoder

After embedding image patches, box tokens and text/function tokens into a common d-dimensional space, we prepend a learned $[\text{CLS}] \in \mathbb{R}^d$ token and fuse them via a single transformer encoder. We concatenate the CLS token, projected image, box and token embeddings into a single sequence as follows:

$$\Big[\underbrace{\text{CLS}}_{\text{cls token}}, \underbrace{v_1,\ldots,v_P}_{\text{image}}, \underbrace{b_1,\ldots,b_{10}}_{\text{boxes}}, \underbrace{e_1,e_2,e_3}_{\text{text}}\Big] \in \mathbb{R}^{(1+P+10+3)\times d}.$$

We also build a boolean mask $M \in \{0,1\}^{(1+P+10+3)}$ (where P is the number of patches of image features) with zeros for any padded box/text slots and ones elsewhere. The transformer encoder follows the standard architecture as described in the original paper (Vaswani et al., 2017), utilising multi-head self-attention and encoder-decoder attention mechanisms. We stack N layers, after which we obtain a hidden representation:

$$H = X^{(N)} \in \mathbb{R}^{L\times d}, where L = (1+P+10+3)$$

whose rows $H_0, H_{1:P}, H_{P+1:P+10}, H_{P+11:P+13}$ are contextualised embeddings for the CLS token, image patches, box slots, and text/function tokens, respectively. These serve as the memory for downstream decoders. The bounding-box decoder cross-attends to all outputs to predict bounding box coordinates, and the token decoder uses the CLS token to generate the final discrete token.

We use 3 layers for the encoder, with 4 attention heads each. Each encoder layer has dimension d=512.

3.4.2.5 Decoder Routing

After the shared encoder produces $H \in \mathbb{R}^{L\times d}$, we extract the embedding of the function token slot, $h_{\text{func}} = H_{P+19} \in \mathbb{R}^d$. We then pass h_{func} through a linear layer with input dimension d, hidden dimension 512, and output dimension 2 to compute logits for the two decoders (bounding box vs. token). We apply softmax to these logits to determine which decoder to use and dispatch H to only that decoder. During training, we add a cross-entropy loss term for routing to the overall loss function.

3.4.2.6 Bounding Box Decoder

We employ a transformer decoder to predict up to $Q = 10$ bounding boxes, each accompanied by a confidence score. The transformer decoder follows the standard architecture introduced in the original paper (Vaswani et al., 2017), with modifications inspired by the DETR model (Carion et al., 2020). Following DETR, we use learned query embeddings and perform all decoding in parallel, since there is no sequential dependency among the boxes.

Because the decoder is permutation-invariant, the N input embeddings must be distinct in order to produce different outputs. These embeddings—referred to as *object queries*—are learned positional encodings that are added to the input of each self-attention layer, analogous to the encoder. The N object queries are transformed into output embeddings by the decoder, and each is then independently decoded into box coordinates by a two-layer multi-layer perceptron (MLP) with ReLU activation and hidden dimension of 512. A sigmoid function is applied element-wise, yielding N final predictions of the form

$$\left[\hat{x}_{\min}, \hat{y}_{\min}, \hat{x}_{\max}, \hat{y}_{\max}, s\right].$$

Here, $\left(\hat{x}_{\min}, \hat{y}_{\min}, \hat{x}_{\max}, \hat{y}_{\max}\right) \in (0,1)^4$ are normalised coordinates, and $s \in (0,1)$ is the *confidence score*. We then filter these predictions by retaining only those with s above a predefined threshold. In our configuration, we use two decoder layers, each of dimension 512 with 4 attention heads, and set $N = 10$ to match the maximum number of boxes.

3.4.2.7 Token Decoder

The token decoder employs the contextualised `[CLS]` token embedding to predict the final discrete symbol. The classification head projects the `[CLS]` embedding through a linear layer of dimension 512, followed by a softmax activation to produce a probability distribution over the symbol vocabulary. Training is performed with a cross-entropy loss against the ground-truth symbol label.

3.4.2.8 Training and Loss Functions

We train the entire Program Executor end-to-end by minimising a sum of three losses: (1) a routing loss to select the correct decoder head, (2) a loss for the bounding box decoder, and (3) a classification loss for the token decoder.

Routing Loss

To determine which branch to activate, the model computes a two-dimensional score vector from the function-token embedding and applies a softmax to obtain the probability of choosing either the box branch or the token branch. We then minimise a binary cross-entropy loss to encourage the correct choice, ensuring higher probability for the true branch and lower for the other.

$$\mathcal{L}_{\text{rout}} = -\left[\mathbf{1}_{[y_{\text{rout}}=\text{box}]} \log p_{\text{box}} + \mathbf{1}_{[y_{\text{rout}}=\text{tok}]} \log p_{\text{tok}}\right].$$

Where:

- p_{box} and p_{tok} are the softmax probabilities of selecting the box or token branch.
- y_{rout} is the true branch ("box" or "tok").
- $\mathbf{1}_{[\cdot]}$ picks the log-probability corresponding to the true branch.

Bounding-Box Loss

The loss we use for the bounding box decoder is similar to that used by DETR (Carion et al., 2020), with a key difference: while DETR predicts bounding boxes and class labels, we predict bounding boxes with an associated confidence score between 0 and 1. We modify the DETR approach by replacing the class prediction components with a confidence score component.

Following DETR's fundamental matching approach, we predict $N = 10$ box-confidence pairs $\{(\hat{b}_i, s_i)\}_{i=1}^{N}$ and match them one-to-one to the $M \leq N$ ground-truth boxes $\{b_j\}_{j=1}^{M}$ via the Hungarian algorithm. The Hungarian algorithm computes a cost matrix between predicted and ground-truth boxes, then finds the one-to-one matching that minimizes the total cost across all N predictions. Specifically, we compute the cost between prediction i and ground-truth j as:

$$C_{i,j} = \underbrace{\lambda_{L1} \|\hat{b}_i - b_j\|_1}_{\text{(A) L1 distance}} + \underbrace{\lambda_{\text{GIoU}}\big(1 - \text{GIoU}(\hat{b}_i, b_j)\big)}_{\text{(B) overlap penalty}} - \underbrace{\lambda_{\text{conf}} \log\big(s_i + \varepsilon\big)}_{\text{(C) confidence bonus}}.$$

The L1 distance term (A) penalises coordinate differences between the predicted box $\hat{b}_i$ and ground-truth box b_j, encouraging predictions that are close to the correct position and dimensions. The overlap penalty term (B) uses the Generalised IoU (Rezatofighi et al., 2019) to penalise poor overlap between boxes, ensuring predicted boxes have good coverage of their corresponding ground-truth targets. The confidence bonus term (C) rewards predictions with higher confidence scores, making it more likely that high-confidence predictions will be matched to ground-truth boxes, with the negative sign converting this into a bonus that reduces the overall cost. The constant ε is added for numerical stability, preventing logarithm of zero issues during training. The hyperparameters λ_{L1}, λ_{GIoU}, and λ_{conf} allow us to balance these three considerations during the matching process.

After we obtain an optimal matching σ, we apply a regression loss combining L1 and Generalized IoU exactly as in DETR:

$$\mathcal{L}_{\text{reg}} = \sum_{(i,j)\in\sigma} \Big[\|\hat{b}_i - b_j\|_1 + \big(1 - \text{GIoU}(\hat{b}_i, b_j)\big)\Big].$$

Differing from DETR, we introduce a confidence loss in place of their class loss:

$$\mathcal{L}_{\text{conf}} = -\sum_{i\in\mathcal{P}} \log\big(s_i + \varepsilon\big) - \sum_{i\notin\mathcal{P}} \log\big(1 - s_i + \varepsilon\big),$$

where $\mathcal{P} = \{i : (i, j) \in \sigma\}$ represents the set of indices for matched queries. This binary cross-entropy loss encourages high confidence scores for predictions matched to ground-truth boxes and low confidence scores for unmatched predictions.

Finally, we combine these objectives into a single bounding-box loss:

$$\mathcal{L}_{\text{box}} = \alpha \mathcal{L}_{\text{reg}} + \beta \mathcal{L}_{\text{conf}}$$

where α and β are hyperparameters that allow us to tune the balance between accurate localization and appropriate confidence scoring.

Token Classification Loss

If the model is routed to output a token, it computes a logit vector over the answer vocabulary. We apply the standard cross-entropy loss on these logits to encourage a high score for the correct token class. Here, y^* denotes the index of the ground-truth answer token in the vocabulary, so $\text{softmax}(\ell_{\text{tok}})_{y^*}$ is the model's predicted probability for the correct token.

$$\mathcal{L}_{\text{tok}} = -\log\big(\text{softmax}(\ell_{\text{tok}})_{y^*}\big).$$

Total loss

The overall loss for each example is the sum of the routing loss plus the appropriate head loss, depending on the ground-truth routing choice:

$$\mathcal{L} = \mathcal{L}_{\text{rout}} + \mathbf{1}_{[y_{\text{rout}}=\text{box}]}\,\mathcal{L}_{\text{box}} + \mathbf{1}_{[y_{\text{rout}}=\text{tok}]}\,\mathcal{L}_{\text{tok}}. \tag{3.3}$$

Optionally, one can scale each component by a positive weight to balance routing accuracy, localization performance, and token prediction quality.

3.5 Baseline

Our baseline model treats the task as one continuous sequence and predicts the program followed by the answer. To account for this, we introduce the following delimiter tokens: <seq>, </seq>, <prg>,</prg>, <ans>, </ans>. A typical target sequence is as follows:

<seq> <prg> program tokens … </prg> <ans> answer token </ans> </seq>

The model uses a standard transformer architecture. For the image encoder, we employ the same approach as our framework. We use ResNet-101 (He et al., 2016) as the backbone to extract image features at the conv4 layer, then 2D positional embeddings are added to these features. The question is encoded with learned word embeddings combined with positional embeddings, as in the original transformer paper. The image and question embeddings are concatenated and passed through 3 transformer encoder layers, followed by 3 transformer decoder layers, each with hidden dimensions of 512. Training is done with teacher-forced cross-entropy over the full sequence (including all special tokens).

3.6 Evaluation

For evaluation, we utilise several metrics including Intersection over Union (IoU), precision, recall, and accuracy.

3.6.1 Intersection over Union

IoU is used to measure overlap between two bounding boxes, and is widely employed in computer vision tasks such as object detection or image segmentation. It is calculated by dividing the area of intersection between two boxes by the area of their union, this is clearly illustrated in Figure 3.5

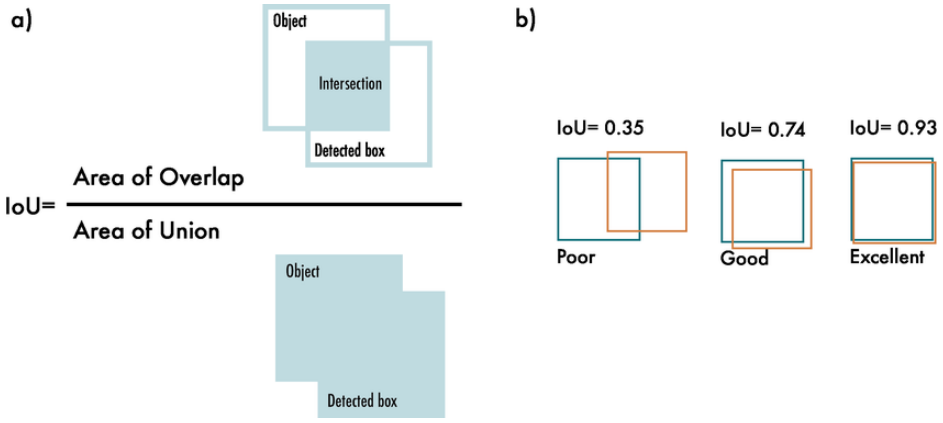


Figure 3.5: Illustration of IoU from Terven et al. (2023).

A higher IoU indicates greater overlap and better performance, with a maximum IoU of 1 representing complete overlap between both bounding boxes. Conversely, a lower IoU signifies less overlap and worse performance, with an IoU of 0 occurring when both bounding boxes completely miss each other.

3.6.2 Precision and Recall

Precision and recall are generally used to evaluate the performance for classification tasks. In the context of our object detection task, these metrics are adapted using IoU thresholds. To understand precision and recall, we make the following definitions:

- **True Positives (TP)**: The model correctly predicts the positive class. In our case, when the model correctly predicts an object with IoU $\geq$ threshold.
- **True Negatives (TN)**: The model correctly predicts the negative class. In our case, when the model correctly predicts no object where none exists.
- **False Positives (FP)**: The model incorrectly predicts the positive class. In our case, when the model predicts an object (or draws a box) where none exists or with IoU $<$threshold.

- **False Negatives (FN)**: The model incorrectly predicts the negative class. In our case, when the model fails to detect an existing object (i.e. misses a ground-truth object or its box has IoU <threshold).

The formulas for precision and recall are then defined as:

$$\text{Precision} = \frac{TP}{TP+FP} \tag{3.4}$$

$$\text{Recall} = \frac{TP}{TP+FN} \tag{3.5}$$

Precision measures the proportion of detected objects that are actually correct, essentially answering "of all the detections made, how many were actually right?" High precision means the model rarely makes false detections. Recall indicates the proportion of actual objects that were correctly detected, answering "of all the objects that should have been detected, how many were found?" High recall means the model rarely misses objects that should be detected.

3.6.3 Accuracy

When running our framework, if it predicts something correctly then it is considered correct. Accuracy is calculated as the number of correct predictions divided by the total number of predictions:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}. \tag{3.6}$$

3.7 Compute Environment

The experiments, which includes training the program generator and program executor on the processed CLEVR dataset, were conducted on Eddie (ECDF), the University of Edinburgh's research compute cluster. Each experiment makes use of a single NVIDIA A100 GPU with 80GB of GPU RAM.

3.8 Summary

This chapter covered several core components for our work:

- We utilise the CLEVR dataset and its CoGenT split. We approximate the bounding boxes using scene graphs and create custom validation and test splits.
- Our framework is made of two models: the Program Generator and the Program Executor. The Program Generator is LSTM-based, whereas the Program Executor is Transformer-based.
- We define a simple Transformer baseline to sue for comparison.
- We detail the loss functions and evaluation metrics used.

Chapter 4

Experiments

In this chapter, we describe the experiments conducted to evaluate our framework against both the baseline and current state-of-the-art models. We carry out three studies: first, we compare performance on the VQA task in terms of accuracy and assess faithfulness through both quantitative metrics and qualitative analysis; second, we examine data efficiency by evaluating the program generator and program executor separately; and third, we test the framework's ability to generalize to new, unseen data.

4.1 Interpretability and Accuracy

4.1.1 Objectives

This experiment captures the main aims of our work. The objectives are two-fold: we want to evaluate our framework based on how it enhances interpretability and faithfulness, specifically how well it grounds the reasoning to the image in our visual question answering (VQA) task, and how the accuracy holds up while achieving these goals. To accomplish this, we will run the full pipeline from question through program generator to program executor and compare final answer accuracy against a baseline and prior work (Yi et al., 2018; Johnson et al., 2017b). We will also evaluate the program generator and program executor individually to better understand their strengths and weaknesses. Additionally, we will determine whether our framework actually improves the visual grounding of this VQA task by analysing whether the intermediate steps of the network meaningfully connect the image to the answer.

4.1.2 Experimental Setup

We begin by training our framework on the preprocessed CLEVR dataset. We train the program generator and the program executor separately as they address different sections of the VQA task.

For the program generator, previous research (Yi et al., 2018) has shown that 9K program examples is sufficient for near-perfect accuracy (for a program generator architecturally

similar to ours). Therefore, we use these 9,000 examples from the training set, utilise 1,000 programs from the validation set, and test on 14,988 programs from the test set.

For the program executor, we utilise the entire preprocessed dataset, making use of all the training, validation and test data.

4.1.2.1 Training Details and Hyperparamters

The main aim of this experiment is not about finding the optimal hyperparameter values, therefore we will only explore a small set of hyperparameters to gain an intuition for their effects on the training of the framework. For both models, we investigated learning rates of 1×10^{-2}, 1×10^{-3}, and 1×10^{-4}, along with batch sizes of 16, 32, and 64. The final configurations yielding the best performance are detailed in Table 4.1, and our baseline model uses the same hyperparameter values as the program executor where applicable.

The loss function described in Equation 3.3 is implemented with weights as specified in Table 4.1 to balance performance across different objectives. For the Program Executor specifically, these weights were assigned to prioritise accurate bounding box predictions while maintaining balanced performance across routing and token classification tasks.

We employ Adam optimisation (Kingma and Ba, 2014) with early stopping to prevent overfitting, halting training when validation loss fails to decrease for consecutive epochs. Visual features are extracted using a frozen pre-trained ResNet-101 backbone from Torchvision (Torchvision maintainers and contributors, 2016). The Program Generator utilises teacher forcing during training to stabilise the learning process.

Hyperparameter	**Program Generator**	**Program Executor**
Learning rate	1×10^{-3}	1×10^{-4}
Batch size	64	16
Training epochs	20	100
Early stopping patience	5 epochs	10 epochs
Dropout	0.3	0.1
Teacher-forcing ratio	0.5	N/A
Routing loss weight	N/A	0.1
Bounding box regression weight	N/A	5.0
Token classification loss weight	N/A	1.0
Bounding box confidence threshold	N/A	0.5
Optimizer	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)	

Table 4.1: Training hyperparameters for the Program Generator and Program Executor

4.1.3 Results and Discussion

4.1.3.1 End-to-End VQA Accuracy

Table 4.2 presents quantitative results for our framework on the CLEVR dataset across all five question types and overall accuracy. We compare our approach with the baseline

model, state-of-the-art (SOTA) models (NS-VQA (Yi et al., 2018) and IEP (Johnson et al., 2017b)), and human performance benchmarks to establish context for our results.

Methods	Count	Exist	Compare Number	Compare Attribute	Query Attribute	Overall
Humans (Johnson et al., 2017a)	86.7	96.6	86.4	96.0	95.0	92.6
IEP (Johnson et al., 2017b)	92.7	97.1	98.7	98.9	98.1	96.9
NS-VQA (Yi et al., 2018)	99.7	99.9	99.9	99.8	99.8	99.8
Baseline	45.8	65.2	51.4	52.9	60.3	54.4
Our framework	69.4	75.1	65.3	64.9	71.1	70.3

Table 4.2: An overview of the accuracy of our framework and other approaches on CLEVR

Our framework achieves an overall accuracy of 70.3%, which represents an improvement over the baseline of 54.4% but falls short of human performance of 92.6% and state-of-the-art approaches such as NS-VQA (99.8%) and IEP (96.9%). Of the individual question types, our framework demonstrates its strongest performance on “Exist" questions (75.1%) while showing comparatively weaker results on “Compare Attribute" questions (64.9%). This suggests varying levels of effectiveness in handling different reasoning steps within the VQA task.

Despite the lower accuracy compared to SOTA models, our framework offers several advantages. It is more flexible and generalisable to new domains without requiring extensive hand-engineering of specialised functions, and also provides interpretability at each processing stage while specifically addressing faithfulness by grounding reasoning to visual elements in the image. Whereas SOTA models like NS-VQA require hand-engineered specialised functions explicitly coded for each reasoning operation, and IEP suffers from interpretability in its decision-making, leaving uncertainty about whether the internal program executions is well connected to the final answer.

The following sections provide a deeper analysis of the individual components of our framework to better understand the specific factors contributing to these performance results and to identify potential areas for improvement.

4.1.3.2 Program Generator Accuracy

Our Program Generator, trained on 9,000 samples, achieves an accuracy of 99.7% on the test set. This high performance aligns with existing literature such as NS-VQA (Johnson et al., 2017b) and IEP Yi et al. (2018), where similar program generators also attain near-perfect accuracy.

Notably, the CLEVR dataset encompasses approximately 450,000 unique programs. Since our model is trained on only a small fraction of these, its performance suggests genuine comprehension of the underlying functional structures rather than just memorisation of specific programs. This outcome is not unexpected, given that questions and their corresponding programs in CLEVR follow a deterministic mapping with well-defined functional composition.

The true challenge in this VQA framework is not about program generation but rather, the visual reasoning capabilities required by the Program Executor, which must correctly interpret and apply these programs to visual content. We will analyse this further in the following section.

4.1.3.3 Program Executor Accuracy

The Program Executor represents the key visual reasoning component of our framework. The executor must handle various functions, some producing single tokens while others generating bounding boxes. We hypothesise that these different output types have different difficulty levels, so we analyse them separately.

Predicting Bounding Boxes

Table 4.3 shows the precision and recall metrics for functions that output bounding boxes. For this analysis, we evaluate each function step independently by using their associated ground truth inputs from our dataset. We consider a predicted bounding box correct if it has an Intersection over Union (IoU) score of at least 0.5 with the ground truth.

Function	Description	Precision	Recall
scene	Return all objects in the scene	0.91	0.93
unique	Return the only object from the input	0.96	0.99
intersect	Return the intersection of two input sets	0.95	0.97
union	Return the union of two input sets	0.96	0.94
same_size	Return all other objects of the same size as the input	0.90	0.89
same_shape	Return all other objects of the same shape as the input	0.85	0.82
same_colour	Return all other objects of the same colour as the input	0.70	0.75
same_material	Return all other objects of the same material as the input	0.79	0.85
filter_size	Return all objects of the specified size from input set	0.93	0.92
filter_shape	Return all objects of the specified shape from input set	0.77	0.82
filter_colour	Return all objects of the specified colour from input set	0.74	0.79
filter_material	Return all objects of the specified material from input set	0.85	0.84
relate	Return all objects of a specified spatial relation to the input object	0.68	0.72

Table 4.3: Per-function box-prediction performance at IoU ≥ 0.5

Our analysis shows clear performance patterns among the different functions. Functions with minimal reasoning requirements—specifically `scene`, `unique`, `intersect`, and `union`—show strong performance with precision and recall above 0.90 in most cases. This makes sense, as these operations mainly manipulate object sets without requiring complex visual reasoning: `scene` simply returns all objects, while `unique`, `intersect`, and `union` perform set operations on the input objects.

In contrast, the `relate` function shows the lowest performance (0.68 precision, 0.72 recall), which we attribute to its complex spatial reasoning requirements. This function

needs to identify objects with specific spatial relationships (e.g., "left of," "in front of") relative to input objects, requiring understanding of three-dimensional relationships and ability to correctly localise bounding boxes.

The attribute-based functions (`same_*` and `filter_*`) fall between the two groups aforementioned. These functions require visual analysis but avoid the spatial complexity of `relate`. Among these, size-based functions (`same_size`: 0.90/0.89, `filter_size`: 0.93/0.92) perform significantly better than other attributes. This is likely because size classification in CLEVR is binary (objects are either "small" or "large"), which directly relates to bounding box dimensions, giving the model helpful cues.

Colour-related functions (`same_colour`: 0.70/0.75, `filter_colour`: 0.74/0.79) show the weakest performance among attribute operations. This probably stems from the greater complexity of colour classification (eight possible colours) and the lack of structural cues like those available for size determination. Shape and material functions perform at intermediate levels, with shape functions generally showing lower precision than material functions.

These performance differences highlight that within bounding box prediction tasks, certain visual reasoning operations are much more challenging than others. The difficulty pattern we observe provides insights for areas of improvement for our architecture, notably the spatial reasoning aspect.

Predicting Single Tokens

Table 4.4 presents the accuracy for functions that output a single token. Notably, these functions demonstrate consistently high performance, with accuracy predominantly in the 0.9 range, approaching perfect performance. This contrasts with the more variable results observed for bounding box prediction functions. We hypothesise that this performance difference stems from the simpler nature of token prediction tasks compared to bounding box localisation. Most token-predicting functions achieve near-perfect accuracy, with only a few showing relatively lower performance.

The query attribute functions (`query_shape`, `query_colour`, `query_material`, and `query_size`) show lower accuracy than other token-prediction operations. This performance gap likely exists because these functions require direct image analysis, whereas other token-prediction functions can derive their outputs by manipulating input tokens without referencing the image. For example, the `equal_attribute` functions simply compare whether input tokens representing object attributes are identical, while counting and existence checks operate on the number of items in input sets.

Among the query attribute functions, `query_colour` shows the lowest accuracy (0.78) while `query_size` demonstrates the highest (0.95). This pattern aligns with our findings for bounding box prediction functions, where colour-based operations were consistently more challenging than size-based ones. As previously noted, this likely occurs because bounding box dimensions provide direct cues for size determination (which is binary: "small" or "large"), whereas colour classification involves eight distinct colours with no comparable structural indicators.

Function	Description	Accuracy
exist	Return whether the input includes any object	0.99
count	Return the number of objects in the input	0.89
query_shape	Return the shape of the input object	0.82
query_color	Return the colour of the input object	0.78
query_material	Return the material of the input object	0.86
query_size	Return the size of the input object	0.95
equal_shape	Return whether the two input shapes are the same	0.97
equal_color	Return whether the two input colours are the same	0.96
equal_material	Return whether the two input materials are the same	0.97
equal_size	Return whether the two input sizes are the same	0.98
equal_integer	Return whether the two input numbers are equal	0.96
greater_than	Return whether the first number is greater than the second	0.94
less_than	Return whether the first number is less than the second	0.98

Table 4.4: Token output functions with descriptions and accuracy

The numerical comparison functions (`equal_integer`, `greater_than`, and `less_than`) all show high accuracy (0.94-0.98). Again, these functions operate on input tokens without requiring image grounding, thus reasonably achieving a very high accuracy.

4.1.3.4 Interpretability and faithfulness

Quantitative Analysis

To analyse the interpretability and faithfulness of our framework, we examine how meaningfully the intermediate steps connect the image and reasoning processes to the final answer. Our analysis categorises outcomes based on program pathway accuracy and answer correctness.

We evaluate each individual program step, considering bounding box outputs correct if they produce all the bounding boxes with an IOU threshold of 0.5, and token outputs correct if they match exactly. An “incorrect program" indicates that outputs from the program executor are wrong for a particular step (causing subsequent steps to fail), rather than an issue with the program sequence itself, as our program generator predicts correct sequences with 99% accuracy.

Table 4.5 compares our framework against a baseline model, showing the distribution of test questions across four outcome categories: reasoning-answer alignment (correct program leading to correct answers), reasoning failures (correct program with incorrect answers), complete failures (incorrect program with incorrect answers), and lucky guesses (incorrect program with correct answers).

Program	Answer	Baseline	Our Framework
Correct	Correct	0.42	0.65
Correct	Incorrect	0.35	0.04
Incorrect	Incorrect	0.01	0.26
Incorrect	Correct	0.12	0.05

Table 4.5: Evaluation of Baseline vs. Our Framework Across Question–Answer Outcomes

The baseline model shows that while it achieves reasonable success with correct programs and answers (42%), it frequently produces misaligned outcomes - either correct programs leading to wrong answers (35%) or incorrect programs somehow yielding correct answers (12%). This suggests a weak connection between the program and final output.

Our framework demonstrates significantly stronger reasoning-answer coupling, with 65% of cases showing both correct programs and correct answers (compared to 42% for the baseline). Importantly, we observe a substantial reduction in misaligned outcomes - only 4% show correct programs but incorrect answers (versus 35% in the baseline), and only 5% show incorrect programs but correct answers (versus 12% in the baseline). When our model's reasoning process fails, the answer typically fails as well (26% of cases compared to 1% in the baseline), confirming that our framework's answers genuinely depend on the reasoning process rather than relying on shortcuts or other correlations.

Qualitative Analysis

our framework provides transparent reasoning paths that can be visually inspected at each step. In cases where the model produces incorrect answers, the execution traces help identify exactly where the reasoning breaks down. For instance, in questions requiring attribute comparison, one can observe how the model first localises the relevant objects, then extracts their attributes, and finally performs the comparison. This step-by-step visualisation makes the reasoning process transparent and verifiable.

To further evaluate interpretability and faithfulness, we conduct a qualitative analysis of representative examples from three informative categories: (1) correct program with incorrect answer, (2) incorrect program with incorrect answer, and (3) incorrect program with correct answer. We exclude the category of correct program with correct answer since it merely demonstrates successful execution where all program steps correctly lead to the correct answer, offering limited additional insights.

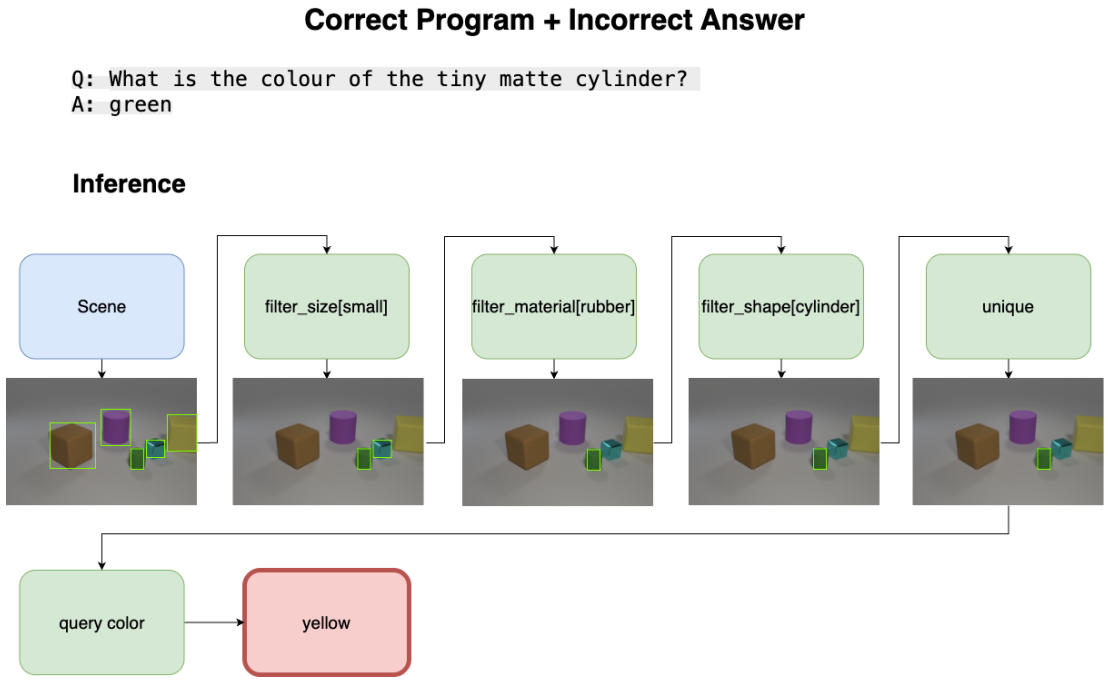


Figure 4.1: Example of correct program with incorrect answer: The program execution is correct until the final step where a colour misclassification occurs.

Figure 4.1 illustrates a case of correct program with incorrect answer. All program steps execute correctly except the final one, resulting in an incorrect output. The error appears to stem from misclassification of the object's colour. This aligns with our earlier findings in Table 4.4, where the `query_color` function exhibited the lowest accuracy among all functions.

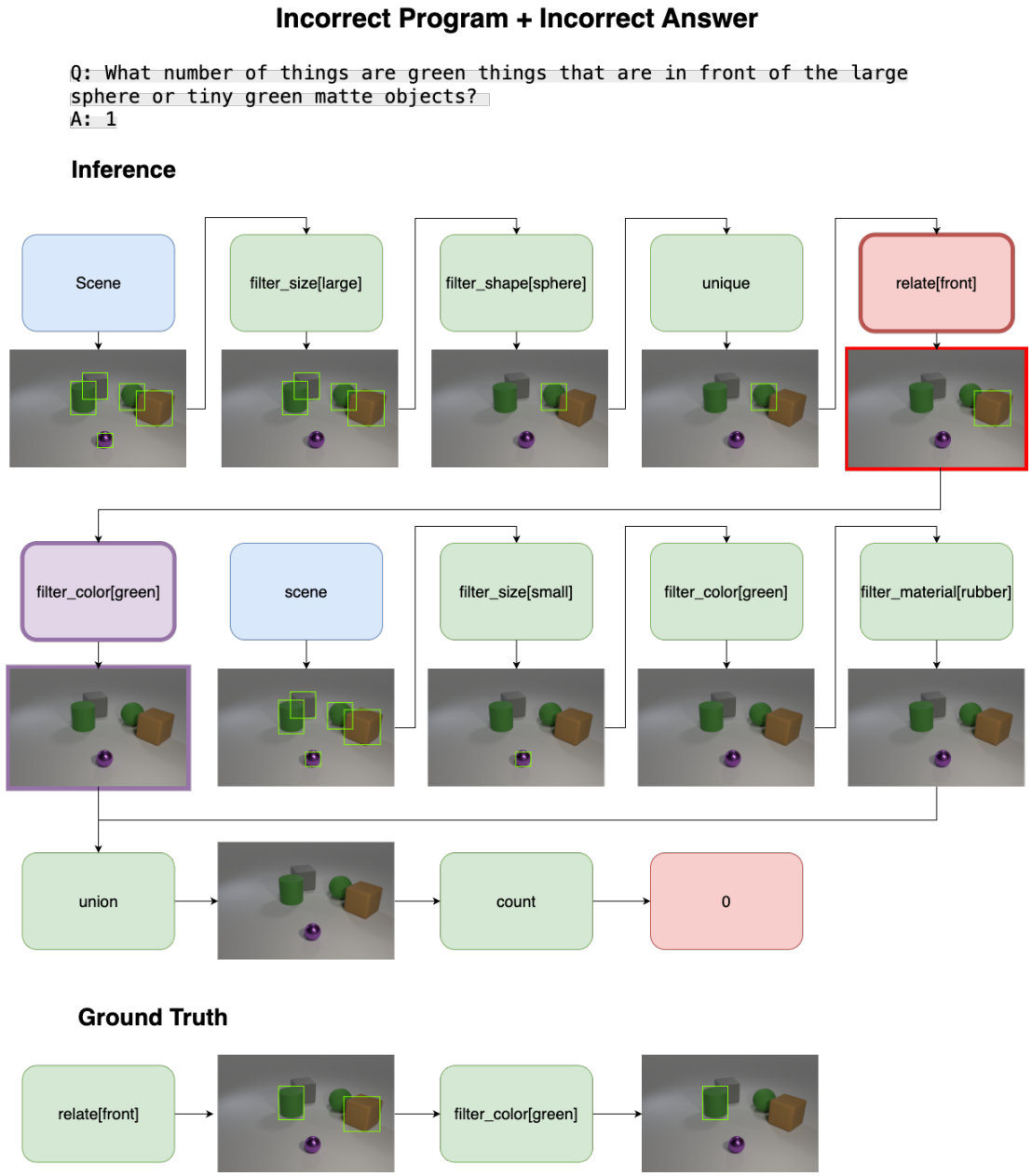


Figure 4.2: Example of incorrect program with incorrect answer: An error in the `relate[front]` function propagates through subsequent steps

Figure 4.2 demonstrates a case of incorrect program with incorrect answer. The `relate[front]` function (highlighted in red) incorrectly identifies objects, which affects the subsequent `filter_color[green]` function (highlighted in purple). Since there are no green objects in the input set the `filter_color[green]` function returns nothing. When this result is combined with the output from the other reasoning branch, the result is an empty set. The `count` function correctly identifies this as containing 0 objects.

According to the ground truth, the `relate[front]` function should have identified two objects rather than one, this misidentified object is the green cylinder. With both objects

correctly identified, the `filter_green` function would have isolated the green cylinder, yielding the correct answer of there being one green thing that is in front of the large sphere or tiny green matte object.

This example effectively demonstrates that when the program's reasoning is incorrect, the resulting answer is correspondingly incorrect, confirming a meaningful relationship between program execution and output answer.

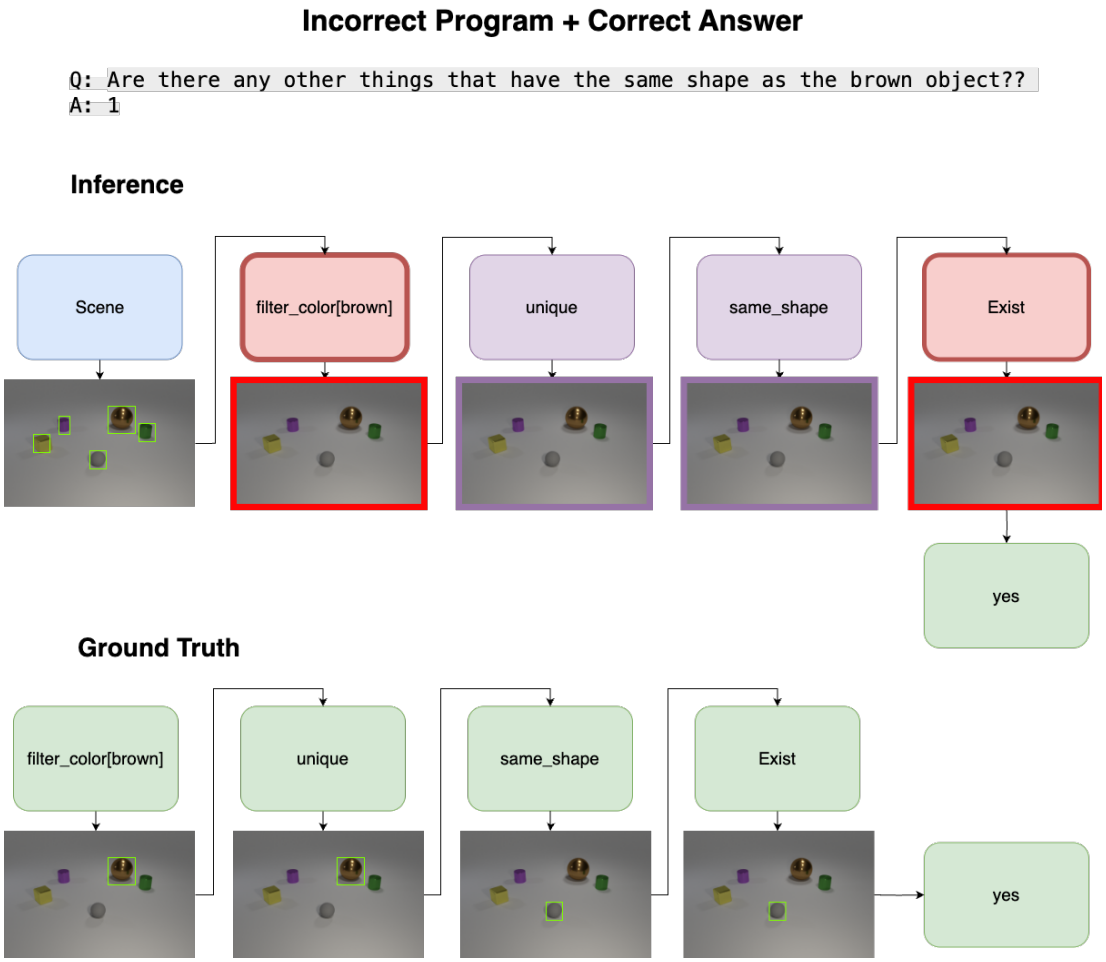


Figure 4.3: Example of incorrect program with correct answer: Despite errors in intermediate steps, the final output is accidentally correct.

Figure 4.3 shows an interesting case of incorrect program but correct answer. The `filter_color[brown]` function incorrectly returns no objects when it should have identified one brown sphere in the upper left. Consequently, all subsequent steps process empty inputs and should logically produce an incorrect final answer. However, the final step surprisingly yields the correct answer. This appears to be due to an error in the last step where the model predicted "yes" instead of "no." This mistake likely occurred because the model has associated the `exist` function with a binary yes/no output and selected incorrectly from these options, inadvertently arriving at the correct answer through faulty reasoning. This behaviour is not desirable, however as shown in Table 4.5, this happens very rarely.

4.2 Data Efficiency

4.2.1 Objectives

The goal of this experiment is to quantify how our framework's performance scales with varying amounts of data. In particular, we aim to investigate data efficiency for the Program Generator, and then, assuming the best Program Generator, investigate data efficiency for the Program Executor.

4.2.2 Experimental Setup

As with Experiment 4.1, we train the Program Generator and Program Executor separately on the preprocessed CLEVR dataset. We use the same hyperparameter settings, as defined in Table 4.1. The difference is that for the Program Generator, we use a varying number of ground truth programs for training: (500, 1000, 9000). For the Program Executor, we use the Program Generator trained with 9000 ground truth programs and vary the number of training questions (7000, 70000, 700000).

4.2.3 Results and Discussion

Figure 4.4 shows results for program generator and program executor accuracy given varying amounts of training data.

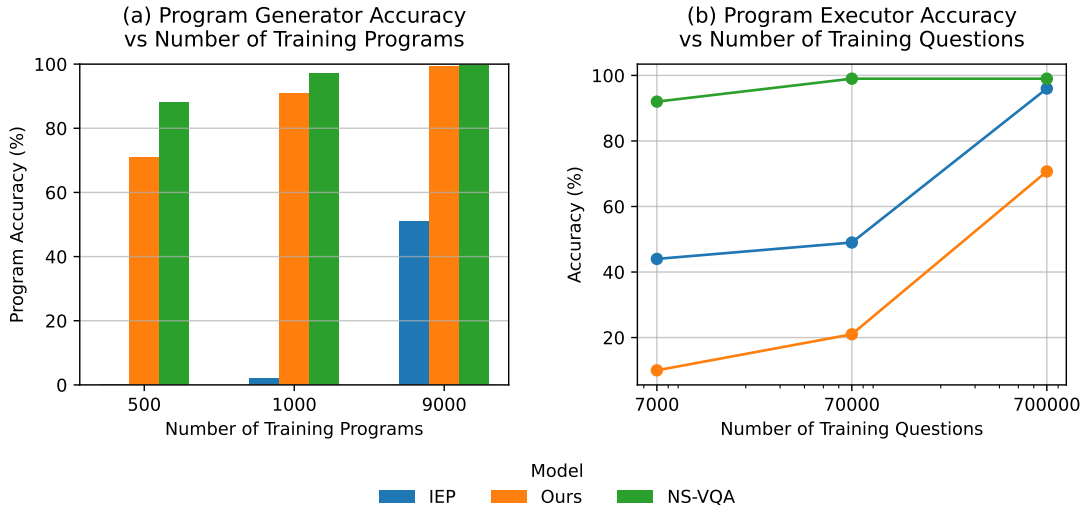


Figure 4.4: Program Generator and Program Executor Accuracy with varying training data.

From plot (a), we see that our Program Generator is very data efficient, achieving good program accuracy with as few as 500 programs. This is in line with NS-VQA's (Yi et al., 2018) Program Generator, whereas IEP (Johnson et al., 2017b) performs notably worse.

From plot (b), it is evident that both NS-VQA and IEP consistently outperform our model across all training data volumes. With 7,000 questions, our model's accuracy

remains below 10%, while IEP reaches approximately 40%, and NS-VQA already surpasses 90%. At 70,000 questions, our model shows modest gains comparable to IEP, while NS-VQA nears perfect accuracy. When trained on 700,000 questions, our model's accuracy rises significantly to just over 70%, yet it still lags behind the near-perfect performance of the other methods.

These results indicate that our approach requires significantly more data to achieve competitive performance. This may be attributed to two main factors: first, our transformer-based architecture inherently demands more data than IEP's CNN-based design or NS-VQA's execution of explicit Python programs; and second, the increased complexity of our task, including the need to predict bounding boxes at each step, which imposes a higher data requirement.

4.3 Generalisation

4.3.1 Objectives

Although approaches like NS-VQA and IEP achieve impressive performance on the original CLEVR dataset, they still exhibit notable deficits in compositional generalisation. In this experiment, we aim to assess the robustness of our framework using the CLEVR-CoGenT dataset. Recall that CoGenT is split into two disjoint sets, A and B, where the allowable colors for cubes and cylinders are swapped between splits.

4.3.2 Experimental Setup

We train our program generator and program executor on the preprocessed CLEVR-CoGenT data. First, both components are trained exclusively on split A: the generator uses 9,000 training programs and 1,000 validation programs, while the executor is trained on all training and validation samples. We then evaluate the framework on the A and B test sets.

Next, we fine-tune the executor on a subset of split B, following the protocol used by NS-VQA and IEP: 3,000 images with 30,000 associated questions. After fine-tuning, we again report accuracies on the test sets of both A and B.

4.3.3 Results and Discussion

Table 4.6 summarises our findings. As expected, IEP and NS-VQA maintain near-perfect accuracy when trained and evaluated on the same split (A→A: 96.6% and 99.8%, respectively). However, in the zero-shot A→B setting, their performance drops dramatically (IEP: 73.7%, NS-VQA: 63.9%), indicating a strong reliance on the color–shape correlations present in CoGenT-A. Fine-tuning on B largely restores their performance (A→B after fine-tuning: 92.7% for IEP, 98.9% for NS-VQA)

In contrast, our framework achieves a more balanced trade-off between in-domain performance and cross-domain robustness. While its A→A accuracy (71.3%) is lower—largely due to the additional challenge of learning bounding-box proposals

from approximate scene-graph annotations—it suffers a smaller relative drop when evaluated zero-shot on split B (60.3%, a 15.3-point decrease). By comparison, IEP and NS-VQA see drops of 22.9 and 35.9 points, respectively. After fine-tuning on B, our model's accuracy on B (64.2%) remains close to its in-domain B performance (66.7%), demonstrating a steadier reasoning pipeline.

We hypothesise that this stability is due to our per-step grounding mechanism. Although IEP and NS-VQA decompose questions into modular reasoning steps, their modules all draw from a shared feature map or an oracle scene graph—never forcing the model to indicate precise image regions. In our design, however, every program step must predict bounding boxes and is supervised directly on those proposals. As a result, each attribute detector (color, shape, material) learns independently of spurious co-occurrences, and every reasoning step is explicitly anchored to the image, minimising the shortcuts that ungrounded systems exploit.

Methods	Not Fine-tuned		Fine-tuned on B	
	A	B	A	B
IEP	96.6	73.7	76.1	92.7
NS-VQA	99.8	63.9	64.9	98.9
Our Framework	71.3	60.3	66.7	64.2

Table 4.6: Generalization results on CLEVR-CoGenT.

4.4 Summary

This chapter examined the performance and properties of our framework through three experiments. We conducted the following studies and obtained these key findings:

- **Experiment 1 (Interpretability and Accuracy):** Our framework achieved a CLEVR accuracy of 70.3%, improving over the 54.4% baseline while offering full transparency into its reasoning steps. We observed strong reasoning–answer alignment (65% of cases with correct program and correct answer) and large reductions in misaligned outcomes (only 4% correct program but wrong answer, vs. 35% for the baseline). Analysis of failure modes highlighted colour classification and spatial reasoning as primary bottlenecks.
- **Experiment 2 (Data Efficiency):** The Program Generator is highly data efficient, reaching near perfect accuracy with as few as 500 training programs. In contrast, the Program Executor required orders of magnitude more training questions to approach competitive performance, achieving just over 70% accuracy only when trained on 700,000 questions.
- **Experiment 3 (Generalisation):** On the CLEVR-CoGenT compositional split, our framework produced a smaller zero-shot performance drop from Split A to B (15.3 points) compared to IEP (22.9 points) and NS-VQA (35.9 points). After fine-tuning on B, our model maintained steady accuracy (66.7% $\rightarrow$ 64.2%), demonstrating enhanced robustness to novel colour–shape combinations.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This work presented a novel framework for spatially grounded compositional reasoning in visual question answering (VQA). Our key innovation is the requirement that reasoning steps explicitly predict bounding boxes around relevant objects in the image, creating a direct and verifiable link between reasoning operations and visual evidence.

Our quantitative analysis for faithfulness revealed a strong correlation between reasoning correctness and answer correctness. When our model's reasoning process succeeds and the answer is correct (65% of cases), and when the reasoning fails the answer in correct (26% of cases). This alignment is substantially stronger than in our baseline model, confirming that our framework's answers genuinely depend on its reasoning path and the image rather than exploiting shortcuts or statistical biases—a key indicator of faithfulness.

Our qualitative examples further demonstrated the interpretability benefits of our approach. By visualising the bounding boxes predicted at each reasoning step, we provided concrete evidence of which image regions influenced each operation. This transparency allows users to verify whether the model is examining relevant parts of the image when making decisions.

While our overall accuracy (70.3%) does not match state-of-the-art methods like NS-VQA (99.8%) or IEP (96.9%), our framework shows better robustness to compositional shifts in the CLEVR-CoGenT dataset. When trained on set A and evaluated on set B, our model experienced a smaller relative performance drop (15.3 percentage points) compared to IEP (22.9 points) and NS-VQA (35.9 points). This suggests that forcing models to ground reasoning steps visually might discourage the learning of spurious correlations.

Finally, our analysis identified clear patterns in function performance. Operations requiring minimal reasoning (like set manipulations) achieved high precision and recall (>0.90), while those demanding complex spatial reasoning (like the "relate" function) performed significantly worse (0.68 precision, 0.72 recall). Similarly, size-based functions consistently outperformed colour-based ones, revealing that certain

visual attributes are more challenging to reason about than others.

In summary, our spatially grounded compositional reasoning framework addresses a key limitation in current VQA systems by enforcing visual grounding at each reasoning step. While not achieving state-of-the-art accuracy, our approach demonstrates good generalisation to novel attribute combinations and provides a faithful, interpretable reasoning process.

5.2 Limitations and Future Work

Given the performance of our framework, several limitations remain and open up opportunities for future research:

- A significant limitation of our approach is that it requires explicit program steps in the training data. However, most real-world VQA datasets lack these fine-grained reasoning annotations. Future work should explore developing mechanisms to generate programs or reasoning steps from scratch without requiring annotated programs for training. Following approaches like ViperGPT (Surís et al., 2023), future work could leverage large language models to automatically generate reasoning programs from natural language questions. This would enable our spatially grounded reasoning approach to extend beyond datasets with program annotations, making it applicable to real-world scenarios where such annotations are unavailable.
- Our current implementation uses a single-pass inference process, meaning that if one step fails, the rest of the reasoning chain is highly likely to fail as well. This creates a cascade of errors that significantly impacts overall performance. A promising direction would be to develop retry mechanisms that allow the model to determine if it has failed at a particular step (perhaps using confidence scores or uncertainty estimates). If the model detects low confidence for a particular operation, it could retry that step until it achieves higher confidence before proceeding to subsequent reasoning steps. This could improve accuracy by preventing error propagation through the reasoning chain.
- Our current implementation uses a relatively simple architecture with limited depth (3 encoder and 3 decoder layers). More powerful models like MDETR (Kamath et al., 2021) utilise up to 12 layers each to achieve better performance. A systematic study on how model scale affects performance would be valuable, potentially revealing the optimal trade-off between computational efficiency and reasoning capacity.

Bibliography

Aishwarya Agrawal, Dhruv Batra, and Devi Parikh. Analyzing the behavior of visual question answering models. *arXiv preprint arXiv:1606.07356*, 2016.

Peter Anderson, Xiaodong He, Chris Buehler, Damien Teney, Mark Johnson, Stephen Gould, and Lei Zhang. Bottom-up and top-down attention for image captioning and visual question answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6077–6086, 2018.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Kishor Barasu Bhangale and K Mohanaprasad. A review on speech processing using machine learning paradigm. *International Journal of Speech Technology*, 24(2): 367–388, 2021.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers, 2020. URL `https://arxiv.org/abs/2005.12872`.

Marie-Catherine De Marneffe and Christopher D Manning. The stanford typed de-

pendencies representation. In *Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation*, pages 1–8, 2008.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL https://arxiv.org/abs/2010.11929.

ECDF. Edinburgh compute and data facility. www.ecdf.ed.ac.uk. Accessed: 15 April 2025.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2(11):665–673, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Michael Hind, Dennis Wei, Murray Campbell, Noel CF Codella, Amit Dhurandhar, Aleksandra Mojsilović, Karthikeyan Natesan Ramamurthy, and Kush R Varshney. Ted: Teaching ai to explain its decisions. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 123–129, 2019.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Xinyu Huang, Jun Tang, and Yongming Shen. Long time series of ocean wave prediction based on patchtst model. *Ocean Engineering*, 301:117572, 2024.

Drew A Hudson and Christopher D Manning. Compositional attention networks for machine reasoning. *arXiv preprint arXiv:1803.03067*, 2018.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2901–2910, 2017a.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *ICCV*, 2017b.

Kushal Kafle and Christopher Kanan. Visual question answering: Datasets, algorithms, and future challenges. *Computer Vision and Image Understanding*, 163:3–20, 2017.

Aishwarya Kamath, Mannat Singh, Yann LeCun, Gabriel Synnaeve, Ishan Misra, and Nicolas Carion. Mdetr-modulated detection for end-to-end multi-modal understanding. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1780–1790, 2021.

Andreas Kamilaris and Francesc X Prenafeta-Boldú. Deep learning in agriculture: A survey. *Computers and electronics in agriculture*, 147:70–90, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Chengen Lai, Shengli Song, Shiqi Meng, Jingyang Li, Sitong Yan, and Guangneng Hu. Towards more faithful natural language explanation using multi-level contrastive learning in vqa, 2023. URL `https://arxiv.org/abs/2312.13594`.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Kun Li, George Vosselman, and Michael Ying Yang. Convincing rationales for visual question answering reasoning, 2025. URL `https://arxiv.org/abs/2402.03896`.

Yibing Liu, Haoliang Li, Yangyang Guo, Chenqi Kong, Jing Li, and Shiqi Wang. Rethinking attention-model explainability through faithfulness violation test. In *International conference on machine learning*, pages 13807–13824. PMLR, 2022.

Scott Lundberg. A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*, 2017.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015. URL `https://arxiv.org/abs/1508.04025`.

Varun Manjunatha, Nirat Saini, and Larry S Davis. Explicit bias discovery in visual question answering models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9562–9571, 2019.

Dong Huk Park, Lisa Anne Hendricks, Zeynep Akata, Anna Rohrbach, Bernt Schiele, Trevor Darrell, and Marcus Rohrbach. Multimodal explanations: Justifying decisions and pointing to the evidence. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8779–8788, 2018.

Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

Protection Regulation. Regulation (eu) 2016/679 of the european parliament and of the council. *Regulation (eu)*, 679:2016, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016. URL `https://arxiv.org/abs/1506.01497`.

Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 658–666, 2019.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Kristof T Schütt, Farhad Arbabzadah, Stefan Chmiela, Klaus R Müller, and Alexandre Tkatchenko. Quantum-chemical insights from deep tensor neural networks. *Nature communications*, 8(1):13890, 2017.

Shahab Shamshirband, Mahdis Fathi, Abdollah Dehzangi, Anthony Theodore Chronopoulos, and Hamid Alinejad-Rokny. A review on deep learning approaches in healthcare systems: Taxonomies, challenges, and open issues. *Journal of Biomedical Informatics*, 113:103627, 2021.

Himanshu Sharma and Anand Singh Jalal. A survey of methods, datasets and evaluation metrics for visual question answering. *Image and Vision Computing*, 116:104327, 2021.

Karen Simonyan. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning, 2023. URL `https://arxiv.org/abs/2303.08128`.

Juan Terven, Diana-Margarita Córdova-Esparza, and Julio-Alejandro Romero-González. A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. *Machine learning and knowledge extraction*, 5(4):1680–1716, 2023.

Torchvision maintainers and contributors. Torchvision: Pytorch's computer vision library. `https://github.com/pytorch/vision`, 2016.

Rakesh Vaideeswaran, Feng Gao, Abhinav Mathur, and Govind Thattai. Towards reasoning-aware explainable vqa. *arXiv preprint arXiv:2211.05190*, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yiming Xu, Lin Chen, Zhongwei Cheng, Lixin Duan, and Jiebo Luo. Open-ended visual question answering by multi-modal domain adaptation. *arXiv preprint arXiv:1911.04058*, 2019.

Seul-Ki Yeom, Philipp Seegerer, Sebastian Lapuschkin, Alexander Binder, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Pruning by explaining: A novel criterion for deep neural network pruning. *Pattern Recognition*, 115:107899, 2021.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Advances in neural information processing systems*, 31, 2018.

Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

Yuke Zhu, Oliver Groth, Michael Bernstein, and Li Fei-Fei. Visual7w: Grounded question answering in images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4995–5004, 2016.

Appendix A

Data Preprocessing

A.1 Bounding Box Approximation

We describe a simple overview of how we compute a 2D bounding box from the scene graph. First, we convert world coordinates into the camera's view. Next, we get an initial size based on how far away and how far to the side the object is. Then we adjust that size depending on the object's shape (cylinder or cube). Finally, we turn our pixel measurements into normalised image coordinates.

A.1.1 Inputs

- **Pixel Coordinates** (x , y , depth) (x,y,depth): The box is centered at (x , y) (x,y) in a 480 × 320 480×320 image. The depth depth value helps us estimate how big the box should be.
- **3D World Coordinates** (X , Y , Z) (X,Y,Z): These give the object's position in the scene. The depth Z Z affects its size, and the side offset Y Y affects perspective.
- **Scene Orientation** ("right" vector): A unit vector (cosine and sine) showing which way is right for the camera. We use it to rotate world coordinates into the camera's view.

A.1.2 Rotate into the Camera Frame

We rotate the object's ground-plane coordinates by the camera's yaw angle. This lines up the object's forward distance and side offset with the camera. Forward distance sets the overall size, and side offset changes apparent width and height.

A.1.3 Initial Size Estimate

We use a constant that converts meters into pixels. Multiplying this by the object's depth and a factor that shrinks with more side offset gives us a base size. We then apply this equally in all directions to make a symmetric box.

A.1.4 Cylinder Adjustment

Cylinders look taller at the top than at the bottom in perspective. So we:

1. Increase the top extent by a factor based on the cylinder's height, depth, and side offset.
2. Recalculate the bottom extent to keep the total height correct.
3. Slightly shrink the left and right extents to match how a cylinder narrows when turned.

A.1.5 Cube Adjustment

Cubes stay square but get smaller as they move to the side. We compute a single scale factor from the side offset and apply it equally to top, bottom, left, and right extents.

A.1.6 Convert to Normalized Coordinates

We find each edge of the box by adding or subtracting the extents from the centre pixel. Then we divide by the image width (480) or height (320) to get values between 0 and 1. Finally, we clamp each number to [0 , 1] [0,1].