Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

# Enhancing the Building Blocks of Language-Based Visual Reasoning for ARC

Master's Thesis


Benjamin Glaus

bglaus@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

**Supervisors:**
Benjamin Estermann, Joël Mathys
Prof. Dr. Roger Wattenhofer


January 3, 2024

# Acknowledgements

I am sincerely grateful to Benjamin Estermann and Joël Mathys for their guidance and support during the course of this thesis. Their expertise has been incredibly useful. I also would like to thank Prof. Roger Wattenhofer and the Distributed Computing Group at ETH Zurich for giving me the opportunity to explore this fascinating topic.

# Abstract

The Abstraction and Reasoning Challenge (ARC) is a benchmark designed to evaluate the general intelligence and human-like cognitive abilities of intelligent systems. Our work builds on top of an architecture introduced by a previous thesis [Camposampiero et al., 2023] that aims at exploiting the prior knowledge embedded in language models. Their architecture consists of a hard-coded captioner, a language model and a decoder.

In this thesis, we aim to further improve the building blocks of this architecture. Similar to the prior knowledge embedded in language models, image captioning models contain vast prior knowledge on the visual domain. We try to exploit this visual prior knowledge for solving ARC tasks by adapting a pre-trained image caption model to act as our captioner. We explore both fine-tuning and reinforcement learning as means to train an image captioning model on producing useful descriptions of ARC tasks. Additionally, we compare the capabilities of multiple language models and test the fine-tuning of a language model for ARC tasks.

# Contents

**Acknowledgements** i

**Abstract** ii

**1 Introduction** 1

**2 Related Work** 3

**3 Preliminaries** 4

3.1 Abstraction and Reasoning Challenge . . . . . 4

3.2 Previous Approach . . . . . 6

3.3 Language Models . . . . . 7

3.4 Image Captioning Models . . . . . 8

3.4.1 BLIP . . . . . 9

3.5 Model Finetuning . . . . . 9

3.5.1 Low-Rank Adaptation . . . . . 10

3.6 Reinforcement Learning . . . . . 10

3.6.1 Proximal Policy Optimization . . . . . 12

3.6.2 Reinforcement Learning from Human Feedback . . . . . 13

**4 Language Model** 15

4.1 Inference . . . . . 15

4.2 Fine-tuning . . . . . 16

4.3 Experiments . . . . . 17

4.3.1 Testing different Language Models . . . . . 17

4.3.2 Error Analysis . . . . . 19

4.3.3 Fine-tuned Language Models . . . . . 21

**5 Learned Captioner** **23**
5.1 Fine-tuning . . . 23
5.2 Data Generation . . . 26
5.3 Reinforcement Learning . . . 27
5.3.1 Environment . . . 28
5.3.2 Actor Critic Architecture . . . 28
5.3.3 Reward Function . . . 29
5.3.4 PPO Training . . . 30
5.4 Experiments . . . 31
5.4.1 Fine-tuning Experiments . . . 31
5.4.2 Reinforcement Learning . . . 32
5.4.3 Inference . . . 32

**6 Conclusion** **37**
6.1 Overview . . . 37
6.2 Future Work . . . 39

**A Language Model** **A-1**
A.1 Parameter Size . . . A-3
A.2 Fine-tuning . . . A-11

**B Captioner** **B-1**
B.1 Fine-tuning . . . B-1
B.2 Dataset Generation . . . B-6
B.3 Reinforcement Learning . . . B-10

Chapter 1

# Introduction

In recent years, the field of artificial intelligence (AI) has witnessed remarkable strides in mastering specialized tasks. More recently, language models and multimodal models started to show the potential of swiftly generalizing to unseen tasks. However, quickly acquiring a new skill given a few examples still poses a challenge. In order to make more deliberate progress towards more intelligent and flexible artificial systems, François Chollet introduced the Abstraction and Reasoning Challenge (ARC) [Chollet, 2019]. ARC is a collection of unique tasks. Each task consists of learning how to transform an input grid into an output grid, given only few examples. It is designed as a benchmark measuring progress towards general artificial intelligence.

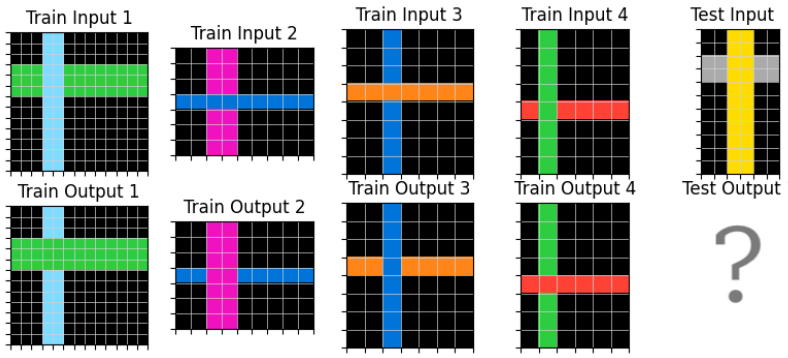



Figure 1.1: Task ba97ae07. Input grids shown on top, with corresponding output grid underneath it. The ordering between the two rectangles or lines is changed. The object initially in the background is moved to the foreground and vice versa.

A system that is able to solve these tasks needs to be familiar with a number of different concepts. For example, an ARC task might require a system to be familiar with occlusion, geometry, sorting, or collisions. Each task is unique in how it is solved and may require a distinct set of prior knowledge.

Figures 1.1 and 1.2 show two tasks from the training set. Each task poses a different challenge and requires a different set of prior knowledge: Figure 1.2 requires

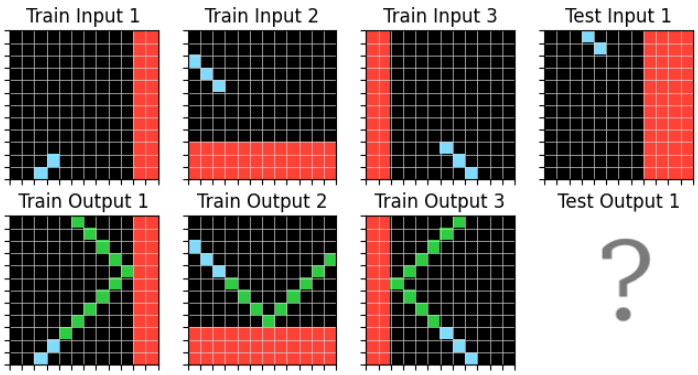



Figure 1.2: Task 508bd3b6. The task consists of extending the diagonal until it collides with the red rectangle and is redirected.

an understanding of physical collisions, and Figure 1.1 requires an understanding of occlusion.

A promising approach to introduce prior knowledge into an artificial system seems to be the use of a Large Language Model (LLM). LLMs are trained on enormous amounts of human-written texts and incorporate vast knowledge in the form of language. While LLMs might are familiar with much of the prior knowledge required to solve ARC tasks, concepts more related to vision, for example occlusion, can pose a problem. Additionally, leveraging the prior knowledge of LLMs to solve ARC tasks, requires the task to be represented as a text (or as a list of tokens).

In this thesis, we expand on a language model based approach to solve ARC tasks. The original architecture consists of a hard-coded captioner, a language model and a decoder. The hard-coded captioner turns a task into a text description of it by following hard-coded rules. Based on this description, a language model is prompted to produce a description of the missing output. Finally, hard-coded decoder is used to turn this description back into a grid.

We aim to build a learned captioner based on an image captioning model. Just like language models have been extensively trained on vast textual data, image captioning models have similarly undergone extensive training on large datasets of images.

By replacing the hard-coded captioner with one based on an image captioning model, we hope to achieve a better understanding of vision related concepts, for example occlusion.

Chapter 2

# Related Work

The Abstraction and Reasoning Challenge (ARC) was introduced by François Chollet [Chollet, 2019] in 2017. It consists of 400 training tasks, 400 validation tasks and 200 secret test tasks. The challenge serves as a benchmark for general artificial intelligence. Most approaches to solve ARC tasks focus on the use of domain specific languages (DSL) and program searches [Fischer et al., 2020], solving up to 30% of the tasks from the private test set. However, a study estimates that humans are able to solve around 80% of ARC tasks [Johnson et al., 2021]. While DSL-based approaches require a hand-crafted set of rules, humans use a much wider range of language when asked to give instructions on how to solve ARC tasks [Acquaviva et al., 2022].

Recent efforts started to include language models in their approach for solving ARC tasks [Xu et al., 2023] [Camposampiero et al., 2023] [Tan and Motani, 2023]. For these approaches, the task first needs to be transformed into a text description, before it can be given as a prompt to the language model. In this work, we focus on building a learned captioner to produce text description of ARC tasks.

Popular image captioning models like BLIP (Bootstrap Language Image Pre-training) [Li et al., 2022] are primarily trained on real-world visual data and struggle with describing more abstract graphics such as ARC tasks. Our aim is to leverage fine-tuning and reinforcement learning methods to use an image captioning model for describing ARC tasks.

Chapter 3

# Preliminaries

In this chapter, we provide a brief overview of notions and topics relevant to our project. We start by introduction ARC, the Abstraction and Reasoning Challenge. Next, we describe the previous approach that was developed during a prior semester project and serves as a starting point of this thesis. We continue with describing language models and image captioning models. Finally, we discuss how these models can be trained using fine-tuning, in particular low-rank adaptation, and how they can be trained using reinforcement learning.

## 3.1 Abstraction and Reasoning Challenge

The Abstraction and Reasoning Challenge (ARC) is a collection of tasks, designed to measure machine intelligence and compare its similarity to human priors. The collection is made up of 400 training tasks, 400 evaluation tasks and 200 test tasks. The test tasks are intentionally undisclosed by the author to prevent bias in the evaluation. Figure 3.1 shows one of the training tasks.

François Chollet introduced ARC [Chollet, 2019] with the purpose of encouraging research in Artificial General Intelligence (AGI) by providing a benchmark for the general intelligence of a system.

The objective of a task is to generate the correct output grid based on an input grid. To learn the correct transformation, each task provides on average 3 training examples. A training example consists of an input grid with the corresponding output grid. Additionally, a task contains one or more test instances of only an input grid.

The objective is to determine the correct output grid for each test instance, by identifying and following the transformation rules used in the provided examples. The required solution method to get from the input grid to the output grid is unique for each task. Figure 1.1 and Figure 1.2 in Chapter 1 showed two examples of how the prior knowledge required to solve a task can vary.

The tasks in the ARC benchmark rely on various concepts and many different as-

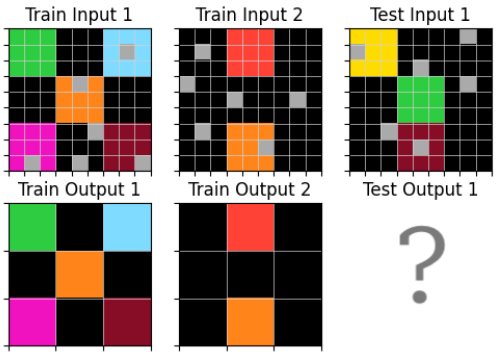



Figure 3.1: Task 5614dbcf. The input grids consist of 9×9 grids containing 3×3 squares in the background, and gray pixels in the foreground. The task consists of removing the gray pixels and scaling the remaining objects down.

pects of knowledge. ARC explicitly groups these knowledge priors into 4 groups:

- Objectness priors: These include the ability to parse a grid into different objects. The shape of an object might be persistent, even if it is partially occluded behind other objects.
- Goal-directedness prior: Many of the input/output pairs can be seen as the starting and endpoint of an intentional process.
- Numbers and Counting priors: include the understanding of numbers and their relations, meaning the ability to count, compare, sort, add and subtract.
- Basic Geometry and Topology priors: The tasks feature a range of elementary geometry and topology concepts. These might be concepts such as lines, shapes, symmetries, reflections, rotations, scales, overlapping, containing, connecting or copying.

When ARC was published in 2017, a competition on Kaggle[1] was held. The winning entry[2] was able to solve 21% of the tasks from a test set. The majority of ARC solutions, including the winning entry, hand-crafted a domain specific language (DSL) of transformations applied to a grid. This approach then searches a sequence of one or more transformations that correctly transforms the input grid into the output grid.

In comparison, [Johnson et al., 2021] looked at human's ability to solve ARC tasks. The study considered a subset of 40 tasks. They found that among 95

[1] https://www.kaggle.com/c/abstraction-and-reasoning-challenge
[2] https://www.kaggle.com/code/icecuber/arc-1st-place-solution

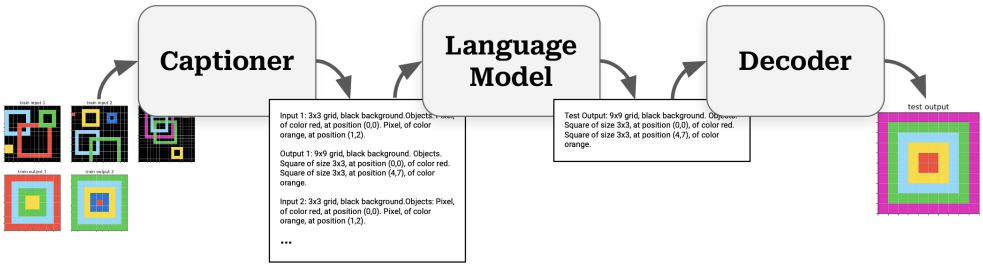



Figure 3.2: ARC tasks are turned into a text description using a captioner. Based on this text description, a language model is prompted to create a text description of the missing output. A decoder turns this description back into an image.

participants, on average humans are able to solve ARC tasks with an accuracy of 83.8%.

## 3.2 Previous Approach

The work presented in this thesis expand on a prior semester project conduced by Giacomo Camposampiero and Loic Houmard [Camposampiero et al., 2023]. In this section, we briefly introduce their thesis.

A central challenge in solving ARC tasks lies in incorporating prior knowledge. The previously described DSL-based approaches make these priors explicit by handcrafting heuristics. Unlike DSL-based approaches, the semester project explored the idea of introducing prior knowledge by including language models into the solution method, as shown in Figure 3.2. Camposampiero and Houmard introduce a framework that consists of first creating a textual description of an ARC task using an encoder. This description is then given to a generic language model. The language model is prompted to predict a description of the task output. Finally, a decoder translates the predicted solution back to a grid.

The captioner in this architecture follows hard-coded rules to produce a description of a task. The visual prior knowledge is still hand-crafted, as the captioner matches objects with a set of hard-coded shapes, including *pixel*, *line*, *square*, *rectangle*, *cross*, *diagonal* or *random object*.

Based on this description, the language model predicts a description of the missing output. Language models are trained on enormous amounts of human-written texts and incorporate large amounts of human knowledge. This prior knowledge may be leverage to solve ARC tasks.

Not only is this approach successful in solving multiple ARC tasks, but it also solved tasks that have not been solved previously by any other approach. From

the training tasks, this approach solved 39 tasks, including 9 tasks that have not been solved by a DSL-based approach.

While their project demonstrated the potential of such an approach, there still are a number of potential improvements. The most promising improvement seems to be a learned approach for the encoder producing the task descriptions. In this work, we will build on top of their work and implement a number of potential improvements.

## 3.3 Language Models

A language model is a statistical or probabilistic model used in natural language processing (NLP) to estimate the likelihood of a sequence of words or characters from a given training dataset. It is designed to capture the syntactic, semantic, and contextual relationships between words within a text corpus.

Language models have found extensive applications in machine translation, sentiment analysis, text generation, and various other NLP tasks due to their ability to understand and generate human-like text.

Language models work by estimating the probability of a token occurring within a sequence of tokens. **Markov models** are a class of probabilistic models that assume we can predict the next word based on only the most recent previous words. This assumption is formulized by the Markov Assumption:

$$P(w_n|w_1 \dots w_{n-1}) \approx P(w_n|w_{n-N+1} \dots w_{n-1})$$

**N-gram models** is a markov model that predicts the next word in a sequence based on the occurrence frequencies of N-grams (sequences of N words).

Neural language models use a neural network as a probabilistic classifier, to compute the probability of the next word. **Recurrent Neural Networks** (RNNs) retain information from previous time steps by adding recurrent connections, feeding the output back into themselves as input. **Long Short-Term Memory** (LSTM) models better retain long-range dependencies by including memory cells that selectively remember or forget information over long sequences.

First described in 2017, **transformer models** [Vaswani et al., 2017] use an attention mechanism to capture contextual information bidirectionally in a sequence. This architecture has proven to result in better quality results, to be better parallelizable and thus needing less training time, and generalizing well to different tasks.

Transformer models introduced a novel architecture based on the self-attention mechanism. Self-attention allows the model to weigh the importance of different words in a sequence, and thereby enables it to consider the context of a word

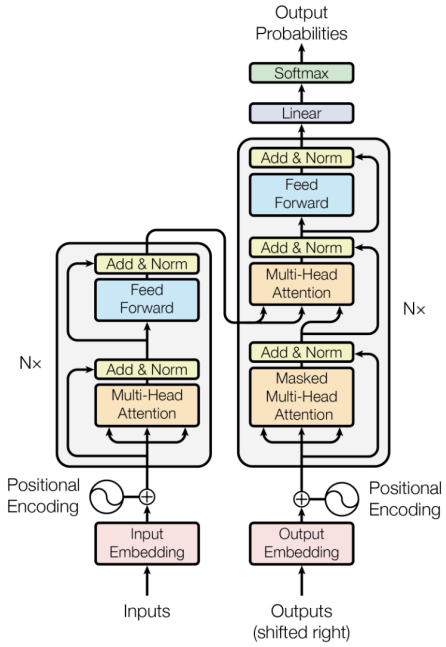



Figure 3.3: The Architecture of a Transformer Model, taken from [Vaswani et al., 2017].

within the entire input sequence. The self-attention mechanism computes attention scores between each pair of words in a sequence. The attention scores are computed through three learned matrices: Query ($Q$), Key ($K$), and Value ($V$). The standard scaled dot-product attention can be written as

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

Figure 3.3 shows the architecture of a transformer model. It consists of an encoder (shown on the left), and a decoder (shown on the right). The encoder maps the input to a continuous embedding. The decoder receives this embedding, as well as the output from the previous step, and generates an output.

## 3.4 Image Captioning Models

Image captioning models aim at creating textual descriptions of a given image. Most image captioning systems use an encoder-decoder framework. An input image is encoded into an intermediate representation, and then decoded into a text description.

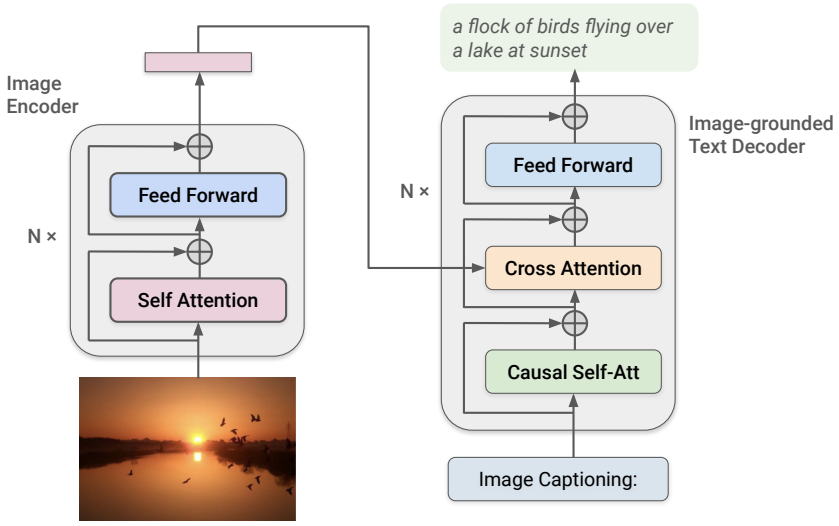



Figure 3.4: BLIP architecture for image captioning.

### 3.4.1 BLIP

BLIP (Bootstrapping Language-Image Pre-training) [Li et al., 2022] is a transformer-based image captioning model. The architecture contains different configurations for image-text matching, visual question answering and image captioning. In the following, we will look at BLIP's architecture for image captioning, as shown in Figure 3.4.

BLIP uses a visual transformer (ViT) [Dosovitskiy et al., 2020] as image encoder, which splits the image into patches and encodes them as a sequence of embeddings. A text decoder initialized from BERT [Devlin et al., 2018] produces a text description of the given image based on the image embedding.

## 3.5 Model Finetuning

When fine-tuning a language model, the model is initialized with pre-trained weights and biases before adjusting model parameters through gradient updates. The model is re-trained on a specific dataset, allowing it to adapt to a specialized task. Running this process the open-source language model Llama 2 [Touvron et al., 2023], with parameter sizes spanning from 7 billion to an impressive 70 billion, demands substantial computational resources. Different strategies exist to make model adaption more compute- and parameter-efficient.

Adapter tuning [Houlsby et al., 2019], in its original design, adds two additional adapter layers per transformer block. During adapter tuning, all pre-trained

weights and biases are kept frozen, and only the adapter layers are updated.

BitFit [Zaken et al., 2021] is a fine-tuning method, where only the bias terms of the language model are modified, while the weights are kept frozen.

More recently, Low-Rank Adaptation (LoRA) [Hu et al., 2021] introduced a parameter-efficient fine-tuning approach that decomposes the weights matrices of a model into low-rank approximations.

### 3.5.1 Low-Rank Adaptation

Low-Rank Adaptation (LoRA) [Hu et al., 2021] freezes the pretrained model weights and injects trainable rank decomposition matrices into each layer of the language model architecture. During training, only the parameters in the rank decomposition matrices are updated. This reduces the number of trainable parameters. For GPT-3, [Hu et al., 2021] reports a 10,000 times reduction of trainable parameters.

For a pretrained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA constrains the update by representing it with a low rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ for the rank $r \ll \min(d, k)$.

During training $W_0$ is frozen, while $A$ and $B$ receive gradient updates. The forward pass can be written as:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

Missing in this formula, LoRA scales the update matrix $\Delta W$ by a scaler $\frac{\alpha}{r}$, where $\alpha$ is a hyperparameter and $r$ is the rank of the low rank decomposition of $\Delta W$. After training, the frozen weight matrix and the low-rank update matrix can be merged, leaving the total number of parameters and the inference latency unchanged compared to the pretrained model.

$$W_{\text{merged}} = W_0 + \frac{\alpha}{r}\Delta W$$

To further improve on the memory efficiency of LoRA during training, QLoRA [Dettmers et al., 2023] is a quantized adaptation of LoRA. The pretrained model is quantized to 4 bits, before LoRA is used to fine-tune it.

## 3.6 Reinforcement Learning

Reinforcement Learning (RL) is a Machine Learning paradigm where an agent learns to make decisions by interacting with an environment. For each action an agent takes, it receives feedback in the form of a reward. The goal is that,

through a process of trial and error, the agent learns a policy that maximizes its cumulative reward.

The environment represents the context in which the agent acts. It might change its state $s$ on its own, or as a result of the agent's actions. The agent sees the environment as observations $o$, which are (partial) descriptions of an environment's state. An actor's policy $\mu$ or $\pi$ decides on the next action, based on an observation: $a_t = \mu(o_t)$ in a deterministic setting, or for a stochastic policy $a_t \sim \pi(\cdot|s_t)$.

The set of all valid actions in an environment is called the action space. Similarly, the set of all possible observations is the observation space.

The reward an agent receives is computed by a reward function $r_t = R(s_t, a_t, s_{t+1})$. The agent's goal is to maximize the cumulative reward across multiple steps. We can denote a sequence of states and actions in an environment as a trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$. The cumulative reward from this trajectory becomes:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

The goal of Reinforcement Learning is to select a policy which maximizes the expected cumulative reward $J(\pi)$:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underset{\tau\sim\pi}{\mathbb{E}} [R(\tau)] = \underset{\pi}{\operatorname{argmax}}\, J(\pi)$$

Many Reinforcement Learning methods are centered around estimating the value function, the action-value function or the advantage function.

The value function $V^\pi(s)$ tells us the estimated total amount of reward an agent will collect in the future, based on the current state $s$:

$$V^\pi(s) = \underset{\tau\sim\pi}{\mathbb{E}} [R(\tau)|s_0 = s]$$

The action-value function $Q^\pi(s, a)$ tells us the estimated total amount of reward an agent will collect in the future, based on the current state $s$ and the action $a$:

$$Q^\pi(s, a) = \underset{\tau\sim\pi}{\mathbb{E}} [R(\tau)|s_0 = s, a_0 = a]$$

The advantage function $A^\pi(s, a)$ describes how much better taking action $a$ in the current state $s$ is, instead of taking the next action based on the policy $\pi$.

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

A family of RL optimization methods that make use of estimations of those functions are the on-policy methods. For these methods, each update to the

policy only uses data collected while actions are taken according to the most recent version of the policy. Since the policy is generating its own training data, an undesirable policy update can cause the next batch of sampled data to be collected from a poor policy. This can cause the optimization process to be unable to recover from a bad policy update.

Trust Region Methods (TRPO) [Schulman et al., 2015] prevent this by constraining the size of the policy optimization by limiting the KL-differgence between the old and new policy. This constraint prevents large changes from a single policy update, ensuring the update to remain within a trustworthy region. However, TRPO requires second order optimization due to the added constraint. This complexity issue was solved by PPO, another on-policy algorithm that allows for first-order optimization, while still limiting the policy updates to stay within a trusted region.

### 3.6.1 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [Schulman et al., 2017] is an on-policy optimization method. Similar to TRPO, PPO limits the policy update to prevent large changes. However, it replaces the constrained used in TRPO with a clipping probability ratios inside the objective function. This allows for first-order optimization, while also constraining the policy updates. Compared to TRPO, PPO is easier to implement and takes less computation time, while keeping TRPO's advantages.

PPO's clipped surrogate objective function takes the minimum between the clipped and non-clipped objective:

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}_t \left[ \min \left( r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{3.1}$$

where $\hat{A}_t$ is an estimator of the advantage function at timestep $t$. In an Actor-Critic architecture, the "Critic" estimates the advantage function. The policy distribution is modelled by the "Actor".

$r_t(\theta)$ denotes the ratio function:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Giving us an easy way to estimate the divergence between the previous policy and the current one. The first part of Equation (3.1), comes from the Conservative Policy Iteration (CPI):

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[ r_t(\theta)\hat{A}_t \right]$$

The second part $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$ clips the ratio function to stay within the interval $[1-\epsilon, 1+\epsilon]$, eliminating the incentive to move $r_t(\theta)$ outside this interval.

The final Actor Critic Objective Function used by PPO adds two more terms to the clipped surrogate objective function from Equation (3.1):

$$L_t^{\text{CLIP+VF+S}}(\theta) = \hat{\mathbb{E}}_t \left[ L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

Where $c_1, c_2$ are coefficients, $S$ is an entropy bonus to ensure sufficient exploration and $L_t^{\text{VF}}$ is a squared-error loss of the value function: $L_t^{\text{VF}}(\theta) = \left( V_\theta(s_t) - V_t^{\text{target}} \right)^2$.

An actor-critic style PPO algorithm is shown here:

**Algorithm 1** Actor-Critic Style PPO

**for** iteration = 1, 2, ... **do**
  **for** actor = 1, 2, ..., N **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  **end for**
  Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
**end for**

During the training process, the actor initially interacts with the environment, gathering experience by following its current policy. Policy refers to the strategy an agent uses to make decisions. During each step of interacting with the environment, the actor collects the initial observation, actions taken, rewards received, and resulting next states.

Based on the collected data, PPO computes an estimate of the advantage. The advantage value describes how much better (or worse) the reward for an action was compared to what was expected in that specific state.

In a second phase, the actor is updated based on the advantage estimates. The probability of taking actions leading to a higher advantage is increased, and the probability of taking worse actions is decreased. The critic is also updated to produce better estimates of the reward the actor will receive.

This process can be repeated multiple times. First, data is collected from interacting with the environment. Then, this data is used to update the model.

### 3.6.2 Reinforcement Learning from Human Feedback

Reinforcement Learning from Human Feedback (RLHF) [Ziegler et al., 2019] is a technique to train a pre-trained Language Model on human feedback through

Proximal Policy Optimization.

The reward an output from a language model receives is computed using a reward model. The reward model should take a sequence of text, and return a reward value, representing human preference. To train this reward model, a set of prompts are given to the initial language model to produce text outputs. These outputs are ranked by human annotators, and the ranking is used to score each output.

The reward model can then be used to train the language model through PPO.

RLHF has been successfully used to train language models for summarization [Stiennon et al., 2020], to browse the web [Nakano et al., 2021] or to better follow instructions [Ziegler et al., 2019].

Chapter 4

# Language Model

In this chapter, we explain our work and experiments related to the language model used in the architecture. First, we describe changes we made to the language model component in the previous approach. Next, we describe how we fine-tuned a pretrained language model on our prompt format. Finally, we show the results from our experiments and briefly discuss them.

As described in Section 3.2, [Camposampiero et al., 2023] implemented a pipeline that solves ARC tasks using a 3-part architecture. A captioner turns tasks into a text description, which is passed to a language model. The language model then tries to predict a text description of the missing output grid, based on the task description. A decoder turns the output description back into a grid. This chapter focuses on the middle part of this pipeline, the language model.

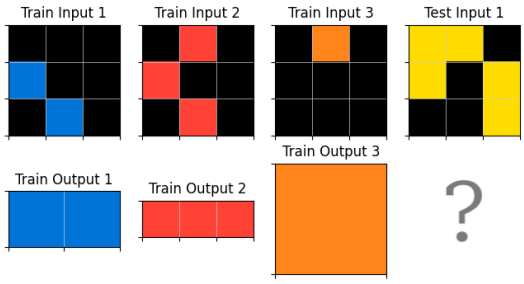
## 4.1 Inference

To evaluate the influence of the pretrained language model on the full architecture's ability to solve ARC tasks, we adapted the code to easily replace the language model. When running inference, the identifier of a text generation model from Hugging Face is passed as a command line argument. The model is then loaded from the Hugging Face Hub, allowing for fast and easy experimentation with different models.

We also adapted the parsing of the language model output, to process the output formats of various language models and extract the description of the predicted output. During Inference, a captioner produces a text description of a task. The task is then formatted to a prompt and passed to the language model. If the prompt is too long for the language model, the task might be skipped.

From the text output, trailing text, the prompt and special tokens are removed in order to extract the predicted description of the task output.

Our prompts follow zero-shot prompting format. Each task is formatted as a single prompt. The language model is given the text description from the training




(a)

Input 1: 3x3 grid, black background. Objects: decreasing diagonal, in position (1,0), of length 2, monochromatic of color blue.
Output 1: 1x2 grid, no background. Objects: horizontal line, with upper left corner in position (0,0), of length 2, monochromatic of color blue.

Input 2: 3x3 grid, black background. Objects: random object of shape 'A', with upper left corner in position (0,0), of size 2x3, monochromatic of color red.
Output 2: 1x3 grid, no background. Objects: horizontal line, with upper left corner in position (0,0), of length 3, monochromatic of color red.

Input 3: 3x3 grid, black background. Objects: pixel, in position (0,1), monochromatic of color orange.
Output 3: 1x1 grid, no background. Objects: pixel, in position (0,0), monochromatic of color orange.

Input 4: 3x3 grid, black background. Objects: random object of shape 'B', with upper left corner in position (0,0), of size 3x3, monochromatic of color yellow.
Output 4:

(b)

Figure 4.1: (a) Task d631b094 and (b) the corresponding prompt passed to the language model.

examples, namely the example input/output grid pairs of the ARC task, together with the test input grid description. The test output grid description is left empty for the language model to predict. Figure 4.1b shows the prompt the language model will receive in order to solve the task shown in Figure 4.1a.

The example input and output grid pairs should enable in-context learning, where we provide demonstrations in the prompt to show the language model how it can solve a task. From these input output pair examples, the language model should infer the transformation needed to produce the correct description of the output grid from the test input grid description.

## 4.2 Fine-tuning

To go beyond the general knowledge of pretrained language models, fine-tuning allows these models to adapt to specific tasks. We implemented LoRA fine-tuning using the PEFT (parameter efficient fine-tuning) library [Mangrulkar et al., 2022] from Hugging Face, allowing for the fine-tuning of any text generation model that can be used for our inference.

We fine-tune the language model on prompts generated for the ARC training tasks. For data augmentation, we allow for the grids and objects within the grids to be reordered.

During training, checkpoints of the model are stored and can later be used for inference.

## 4.3 Experiments

This section outlines two sets of experiments we made related to the language model, and report their results. The first set consists of testing various language models of different parameter sizes. We explore how capable different language model families are in solving ARC tasks, and how the parameter size within a family affects the capability. The second set involves fine-tuning a language model and comparing its performance against a baseline model that underwent no fine-tuning.

### 4.3.1 Testing different Language Models

To find out how much the selection of the language model influences the ability of our pipeline to solve ARC tasks, we tested numerous models. Between all those runs we kept the captioner and decoder fixed.

We tried to select a diverse set of popular language models [1]. We were also interested in seeing how the parameter size of a model affects the result. This led us to include models such as OPT or Bloom, where versions of many different parameter sizes are available.

We report our results for the following language models:

| Name | Parameters |
|---|---|
| Bloom [Workshop et al., 2022] | 500M, 1.1B, 1.7B, 7B |
| OPT [Zhang et al., 2022] | 125M, 350M, 2.7B, 6.7B, 13B |
| OpenLlama [Geng and Liu, 2023] | 3B, 7B, 13B |
| Llama 2 [Touvron et al., 2023] | 7B, 13B, 70B |
| OpenChat [Wang et al., 2023] | 13B |
| Orca Mini [Mathur, 2023] | 3B, 7B, 13B |
| Vicuna [Zheng et al., 2023] | 7B, 13B, 33B |

In Figure 4.2, we show the results on the ARC training tasks. For each of the 400 tasks, we allowed the language model to make 6 guesses. If at least one of those guesses is correct, a task is considered to be solved.

[1] These Experiments were done mainly in July and August 2023. Today, other models might be considered more popular, or newer versions of these model might be available.

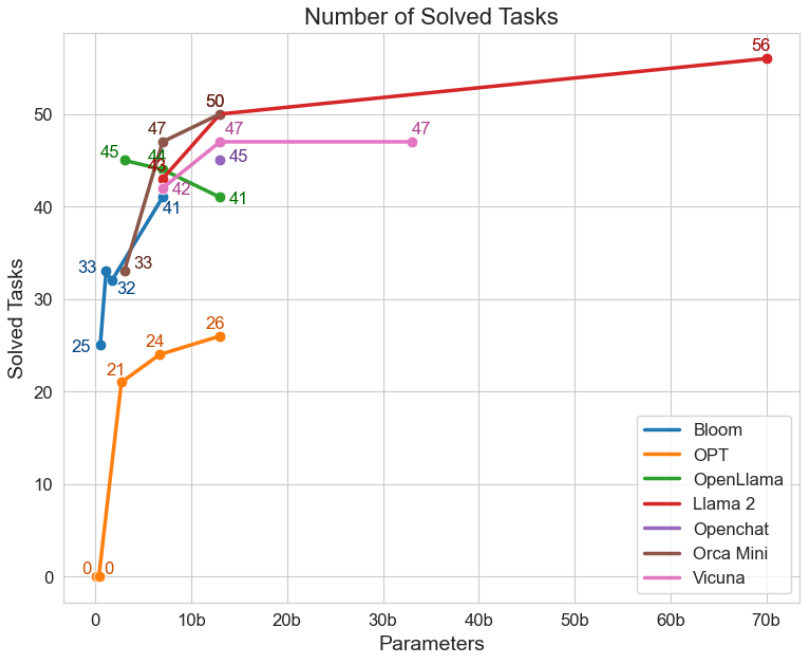



Figure 4.2: Number of training tasks solved by different language models. All models were run with objects sorted by size and the multichromatic captioner with diagonal connections (SMuD).

We observe that with increasing parameter size, the models are able to solve more tasks. In particular for smaller model sizes, between 0 and 10 billion parameters, increasing size leads to significantly more tasks solved. However, this increase in performance diminishes with increasing model size.

For example, when comparing the 3 billion and 7 billion parameters versions of Orca Mini, the number of solved tasks increases from 33 to 47 tasks solved (an increase of 55%). However, the 13 billion parameters version of Orca Mini only solves an additional 3 tasks, leading to a total of 50 tasks solved (an increase of 6.4%).

Another observation we can make is that while many language models perform somewhat similar, there is also some difference in performance. Most notably, the OPT models seem to perform significantly worse than other models.

The most surprising observation comes from the OpenLlama models. With an increase in parameter size, fewer tasks are solved. Looking at the guesses made for a task, we observed that the smaller version made better use of the guesses by producing more diverse output grid descriptions. The larger model was more consistent in producing a description similar to its first guess, potentially leading it to solve fewer tasks. However, it is unclear why this has only been observed for the OpenLlama models.

### 4.3.2 Error Analysis

In order to gain a deeper understanding of our results, we will take a closer look at the answers generated from Llama-2 13B and the errors that occurred. For each task, the language model makes 6 guesses. The language model chooses the top 6 sequences by beam search. Beam search explores multiple possibilities by keeping track of the top 6 sequences at each decoding step and selecting the most probable continuations. Each of these guesses is passed to the decoder. A guess is correct, if the output of the decoder perfectly matches the true test output.

The outcome of each task is shown in Figure 4.3. *Correct Result* contains all the tasks for which at least one guess from our pipeline was correct. *Wrong Result* includes all tasks for which the pipeline produced at least one output grid, but not the correct one. *Failed* includes all tasks for which no grid was produced. Finally, a task is skipped if the prompt produced from the captioner is too long.

For each one of the 400 training tasks, our language model produced 6 guesses. Our decoder failed for 774 out of those 2'400 guesses. Figure 4.4 shows the cause of these failures.

- **Random Object / Patterns:** Most commonly, failures are caused by random objects or multichromatic patterns. We allow the captioner to describe shapes as random objects. These random objects are stored in a

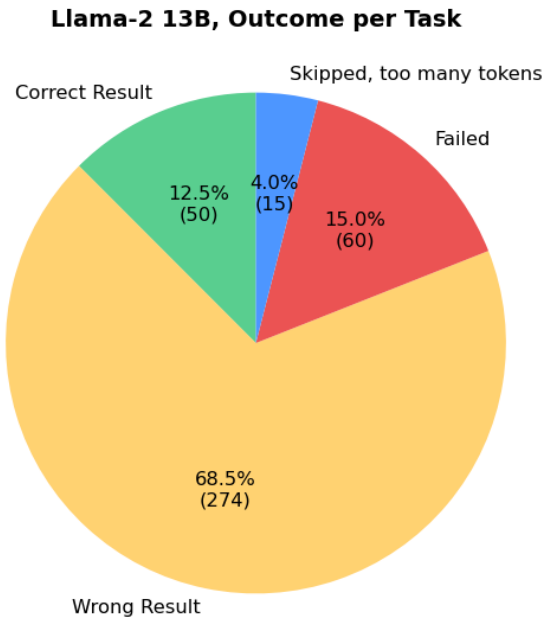



Figure 4.3: Outcome of solving the training tasks using the Llama-2 13B language model.

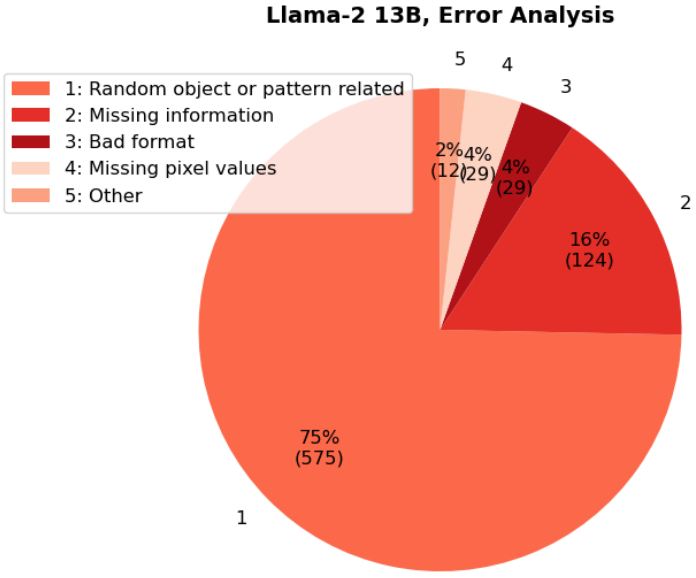



Figure 4.4: Different types of errors observed when using Llama-2 13B.

dictionary and can be referenced by the language model, for example by describing a *random object of shape "B"*.
The decoder can read the corresponding shape from the dictionary. If the dictionary does not contain the specified key (*"B"* in the example above), this causes a failure. In many cases, the language model "learned" to predict the next random object identifier. For example, if the task description contained random objects *"A"*, *"B"*, *"C"* and *"D"*, the language model predicted the non-existing random object *"E"*.
Similarly, the color pattern of multichromatic objects is described as *pattern "a"*, using some lower case letter as identifier. Similar to before, the language model frequently predicted a non-existing pattern.

- **Missing Information:** Another source of failures is the language model omitting necessary information, like the grid size.
- **Bad Format:** The output generated from the language model cannot be parsed by the captioner. For small models, this happens because they struggle to fully follow our description format. For larger models, we observe that they occasionally add reasoning or "chain-of-thought" steps to their response, causing the decoder to be unable to parse the generated text.
- **Missing pixel values:** If the grid is fully covered by objects, the captioner may describe no background color. If the language model predicts the output to not have a background color, it has to describe objects that fully cover the grid, otherwise some pixels are left undefined.
- **Others:** Other failures are mostly caused by the language model specifying colors that do not exist for ARC tasks. Most common is *white*, others include *magenta, pink* or *lightgray*.

Table A.1 in the appendix shows the number of solved, wrong, failed and skipped tasks for all language models we tested. In Appendix A.1 we show which tasks are solved for different size of the Llama-2 model.

### 4.3.3 Fine-tuned Language Models

Based on the results from testing different language models, we selected Llama-2 13B to make additional experiments. We fine-tuned the model on the training tasks of ARC and used the evaluation tasks to compare the performance with the baseline llama-2 13B model without fine-tuning.

The data used for fine-tuning consists of the captions produced for the training tasks. We report the results for two different fine-tuning, differing in the training data. For the first run, *Fine-tuning 1* in Figure 4.5, we used the captions of the 400 training tasks. For each task, we used the captions with the objects sorted

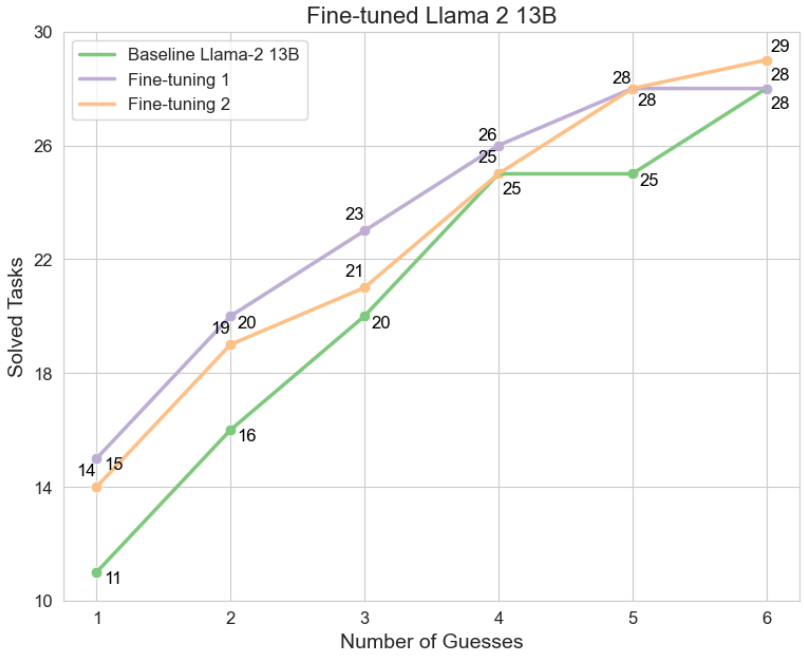



Figure 4.5: Number of tasks solved against number of guesses made by the language model. Comparing Baseline Llama 2 13B with fine-tuned versions.

randomly, by size and from left to right. This gives us a total of 1'200 training examples. For the second run, we augmented those by reordering the objects within a grid, leading to a total of around 12'000 training examples.

Appendix A.2 in the appendix contains additional information about the fine-tuning process and shows which tasks have been solved by the fine-tuned models, compared to the baseline Llama-2 13B model.

Figure 4.5 shows the number of evaluation tasks solved against the number of guesses made by the language model. For example, when only allowing the models to make 3 guesses, the baseline model was able to solve 20 tasks. The two fine-tuned models were able to solve 21 and 23 task respectively within those 3 guesses. The difference between the baseline and the fine-tuned models is the biggest when only allowing 1 or 2 guesses, with the fine-tuned models solving up to 4 task more than the baseline model. However, when allowing more guesses, the models solve almost an identical number of tasks.

Chapter 5

# Learned Captioner

In this chapter, we focus on the first step in our pipeline, the captioner. The captioner is responsible for the text description that is passed to the language model. Since the language model does not receive any representation of the task other than this description, its ability to solve tasks relies heavily on receiving a high-quality task description.

In the following sections, we will describe how we build a learned captioner, based on a pre-trained image captioning model. We first describe how we fine-tune a pre-trained image captioning model on a synthetic dataset of image-caption-pairs resembling ARC tasks. We then take a closer look at how we build this dataset. Afterward, we outline how we continued training the fine-tuned captioner using Reinforcement Learning.

Finally, we conclude this chapter by presenting and discussing our results.

## 5.1 Fine-tuning

The hard-coded captioner produces the text description of a task based on hand-crafted rules. This limits the captioner's ability to deal with challenges such as overlapping shapes, handling noise, distinguishing touching or overlapping shapes of the same color, or recognizing random objects based on occurrence in another grid of the same task. Another issue we frequently observe is that it fails to distinguish between two touching or overlapping shapes, and a single random shape.

Ideally, a captioner should even be able to distinguish the level of abstraction on which it describes a task. While some tasks might require information about every object, other tasks include plenty of noise, meaning pixels or object that do not affect the solution.

Task 7e0986d6, shown in Figure 5.1, is a good example. The pixels in the input images overlap with the rectangles in the background. In the first training input, two rectangles in the lower left corner also touch each other. For a human ob-

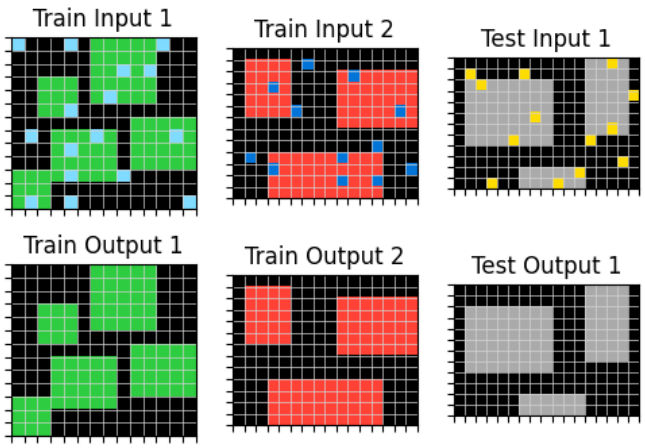



Figure 5.1: Task 7e0986d6 from the ARC training tasks.

server, it is still easy to recognize 5 distinct green rectangles. To solve the task, the level of detail we require for each object is also different. While we do not need any details about the noisy pixels, we do need to know the position, shape, and color of the rectangles in the background.

To make the captioner more versatile, we experimented with BLIP (Bootstrap Language Image Pre-training) [Li et al., 2022], a pre-trained image captioning model. BLIP is pre-trained on a large dataset of images paired with corresponding textual descriptions. This dataset is primarily made up of real-world visual data, leading to the model being largely proficient in understanding and associating real-world visual content with descriptive text.

While it can generalize to some extent and may recognize basic shapes or abstract representations, it struggles to produce useful description for ARC tasks. In Figure 5.2, we show an example of captions produced by a BLIP model without any fine-tuning. Without fine-tuning, the captions produced by BLIP lack plenty of relevant information and most likely will not be useful.

Our goal for fine-tuning BLIP is to create a captioner that can describe objects and grids of an ARC task one-by-one. During each step, the captioner is given an image of the task as an input, and should produce a caption of a single object or grid background as output.

For each grid, we iteratively use the captioner to describe the objects one-by-one. Once all the objects in the grid are described, the captioner should produce a description of the grid background. For example, for a grid containing a single object, it should first output a description of that object:

*pixel, in position (1, 12), of color blue.*

In this example, this is the only object in the grid. The model is done with

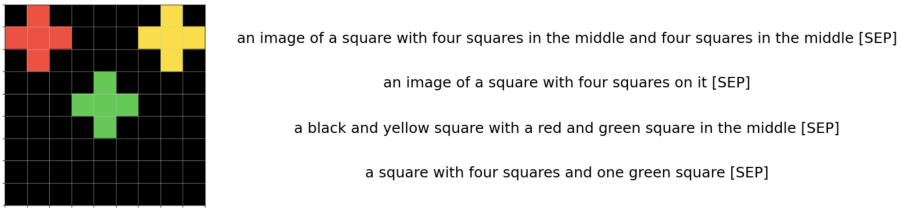



Figure 5.2: Captions produced by BLIP without any fine-tuning. The grid on the left shows a grid from training task b190f7f5.

describing all objects in the grid, and should describe the grid background next:

*15x21 grid, black background.*

After the model described the background, we know that it is done with describing this grid, and can move on to the next grid in the task.

In order for BLIP to know which object to describe next, it needs to know which grids and objects it has described earlier. As mentioned before, the model receives an image of the task as input at every step. In this image, we mask-out objects and grids that have been described previously by brightening their color, as shown in Figure 5.3b. Based on this masking, the model should understand which objects and grids it described previously. In order for the model to know which object to describe next, we teach it to describe them left-to-right, top-to-bottom.

To summarize, we want to use fine-tuning to teach BLIP the following:

- Object description: correctly describing an object by providing all relevant information: position, color, object type, and shape.
- Grid description: correctly describing a grid background by providing all relevant information: color and shape.
- Masking: given a task image, the model should understand which objects and grids have been described previously.
- Description ordering: based on the masked objects, the model should understand which objects it has to describe next. Objects are described left-to-right, top-to-bottom. Once all objects are described, the model should describe the grid background and afterward move on to the next grid.

To teach this to BLIP, we generate a large dataset of image-caption pair examples. For each pair, the caption corresponds to the output we want the model to generate, if it is given the corresponding image as input. We will describe the generation of this dataset in the next section in more detail.

During fine-tuning, we initialize a pre-trained BLIP and fine-tune it on our custom dataset. Compared to a language model, BLIP is relatively small, consisting of around 250 million parameters. Because of its small size, full fine-tuning is feasible, meaning we train all weights and biases. We also experimented with keeping parts of the model frozen, namely the vision encoder.

In Appendix B we share additional information about the fine-tuning process of BLIP. Table B.1 shows the parameters used for fine-tuning. We also added plots showing the color, type, position, and shape accuracy for each of the object types individually.

## 5.2 Data Generation

To create the dataset, we first randomly create tasks consisting of arbitrary grids. These grids may contain objects, of the object types defined in the previous approach: pixels, lines, diagonals, rectangles, squares, crosses or random objects.

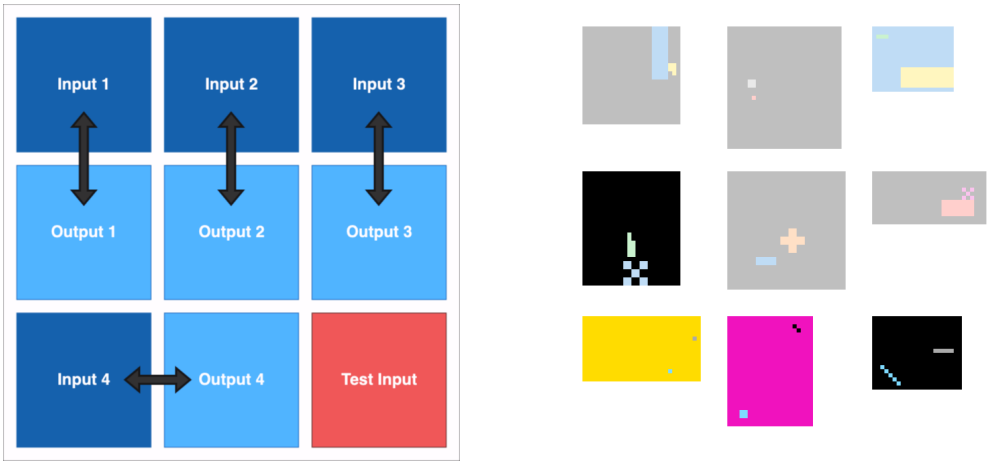
For each task, we iterate over the grids and objects to produce image-caption pairs. The image-caption pair represent the caption we want the captioner to produce for the given image. When producing these pairs, previously described objects get masked out by brightening their color. For example, when describing the second object in the second grid, we mask out the complete first grid, as well as the first object in the second grid. This way, the captioner should be able to understand that it needs to describe the second object next, and create a caption for the next object. For example, this is how a single pixel is described:

*pixel, in position (1, 22), of color blue.*

For the captioner to correctly describe the next object, we need it to understand an ordering in which it describes the objects. When we iterate over the objects in a grid, we do so in a left-to-right, top-to-bottom order. By using this ordering for our dataset and masking out previously described objects, we expect the captioner to understand this ordering, and correctly know which object it has to describe next. Once all the objects in a grid are described, we want the captioner to create a caption for the background next:

*15x21 grid, black background.*

The images are formatted to contain all task grids. We arrange the grids in a 3×3 format. This way, we can show 4 pairs of example input and output pairs, as well as a test input grid. In Figure 5.3a, we show how the grid arrangement is done.




(a) (b)

Figure 5.3: (a) The input and output grids of a grid are arranged in a 3×3 pattern. The test input grid is placed at the bottom right. (b) Previously described objects or grids are masked out by brightening their color.

Figure 5.3b shows an example from the dataset, with previously described grids and objects masked out. In Appendix B.2 in the appendix, we show a full example of how we iterate over the objects and describe them one-by-one. We found that keeping the grid that is currently being described at the top left (in the position of input 1 and input 2), makes it easier for the captioner to get positions and sizes correct. Thus, when creating image-caption pairs, we swap the grids to always keep the currently described grid in those positions.

We specified a number of different hyperparameters that can be used to customize the dataset. These include the grid height, grid width, number of objects within each grid, object types, allowing overlapping objects, number of example grid pairs, and background color.

## 5.3 Reinforcement Learning

By fine-tuning BLIP on our custom synthetic data, we now have a captioner that is capable of describing ARC grids with a high degree of accuracy, though not entirely perfect. All the training data it has seen is synthetic and may vary from how real ARC tasks look like.

ARC provides us with 400 training and validation tasks each. Unlike our synthetic datasets, the dataset of ARC task does not have any matching textual descriptions of the grids. Since it lacks any captions or object annotations, we cannot do any form of supervised training, like we did for our custom dataset. For our captioner to profit from this dataset, we look into Reinforcement Learning.

We use our fine-tuned captioner to create captions for ARC tasks. We specify a reward function that returns a reward for each action the captioner takes: when the captioner produces a full object or grid description, we use the decoder to get the pixels of the described object. Together with the position of the predicted object, we compare the predicted pixels with the true pixels. The reward function computes a reward value based on the number of predicted pixels and true pixels.

Using PPO, we continue training the model on this reward value, hoping that it can improve the quality of the generated captions.

Our Reinforcement Learning setup consists of 4 main components: the environment, an actor, a critic, and the reward function.

### 5.3.1 Environment

The environment is the context with which the actor interacts. It represents the current state and consists of the previously generated tokens, the image representation of the task, the mask of previously described parts of the image, and a processor.

The processor is used to turn the masked images and previously generated tokens into the actor's expected input format, tensors. We call the input for the actor an **observation**, and the set of all valid observations the **observation space**. Similarly, the **action space** is the set of all valid actions the actor can take during a step in the environment.

### 5.3.2 Actor Critic Architecture

The Actor-Critic Architecture implements our actor and critic. Our **Actor** is a fine-tuned BLIP checkpoint. As an input, it takes an observation consisting of 3 tensors representing the current task image and previously generated tokens: pixel values, input ids, and attention mask.

Based on this observation, it decides on the next action to take, meaning which token it should produce next.

The **Critic** receives the same observation and produces an estimate of the reward the actor will receive. As described in the preliminaries, this is an estimate of the advantage function. PPO uses this estimate to limit the policy updates and prevent large changes.

We adapted the architecture of BLIP to function as our critic. We use the same vision model as for our critic, but pass its output to BLIP's text encoder. We take the output from the text encoders last hidden state and pass it to two dense layers, reducing the size from 768 to 264, to 1.

Since we share the vision model between actor and critic, we can reduce computation by only computing its output once per observation, and then passing it to the remaining layers of the actor and critic respectively.

### 5.3.3 Reward Function

The reward function returns a reward value for each action our actor takes. It computes this value based on comparing the true pixels with the predicted pixels described by the caption produces by the actor. When the actor finishes describing an object, we use the decoder to turn the text description into the corresponding pixel values of that object. Then we count the number of pixels described in total, as well as the number of pixels matching the true image. Based on those two values, we designed the following reward function:

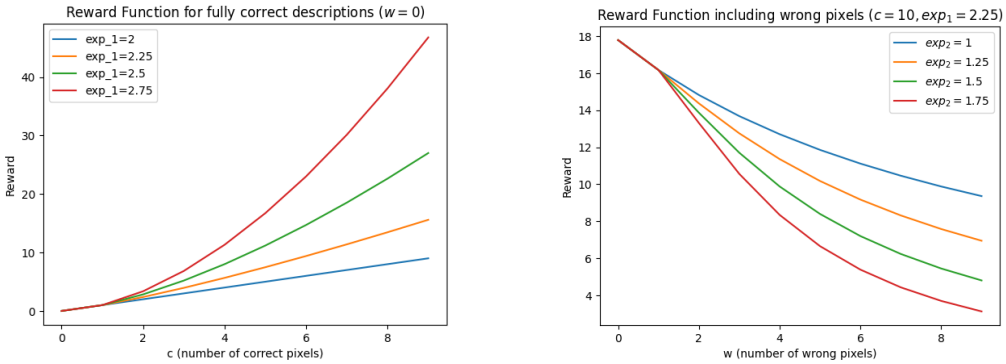
$$\text{reward}(c, w) = \frac{c^{\exp_1}}{c + w^{\exp_2}} \tag{5.1}$$

Where $c$ corresponds to the number of correctly predicted pixels, $w$ to the number of wrongly predicted pixels, and $\exp_1$ and $\exp_2$ are two parameters to change the functions' behavior.

This reward function is designed based on 2 considerations.

- Describing a set of pixels as a single large object instead of multiple smaller ones should yield a larger reward.
  Assuming we have a perfect captioner that does not predict wrong pixels, we have
  $$\text{reward}(c, w) = \text{reward}(c, 0) = \frac{c^{\exp_1}}{c + 0^{\exp_2}} = c^{\exp_1 - 1}$$
  This reward should be larger than the sum of rewards from predicting the same object as two smaller ones:
  $$c_1^{\exp_1 - 1} + c_2^{\exp_1 - 1} < c^{\exp_1 - 1}$$
  Assuming the same amount of pixels ($c = c_1 + c_2$). This holds for $\exp_1 > 1$.
- Adding additional pixels to the object that do not belong to it should not increase the reward. Since our function is designed to give a larger reward to larger objects, it might be possible to add additional pixels to an object to increase the maximal reward the prediction can receive.
  We added a second exponent, $\exp_2$, to control the negative impact wrong predictions have on the reward given.

To prevent the reward values from becoming too large, multiplying the denominator with the nominator's exponent $\exp_1$ can scale it appropriately. We also




(a) (b)

Figure 5.4: Plots based on the reward function: $\text{reward}(c, w) = \frac{c^{\text{exp}_1}}{c+w^{\text{exp}_2}}$. (a) shows the reward given for fully correct descriptions of different sizes. (b) shows the decrease in reward when adding wrong pixels to a description of an object of 10 (correct) pixels.

tested adding a small negative reward at every step to incentivize fewer actions, meaning shorter descriptions.

Figure 5.4a shows the reward given for fully correct objects descriptions of different sizes. The different lines show the effect of different exponent values. Assuming $\text{exp}_1 \geq 2$, we see that with increasing exponent, the reward given for larger objects increases much more than for smaller objects. By giving a larger reward to larger objects, we want to prevent our model from describing objects as multiple smaller ones.

In Figure 5.4b, we see the decrease in reward when adding wrong pixels to an object of size 10. The different lines show how increasing $\text{exp}_2$ causes the reward to decrease faster.

### 5.3.4 PPO Training

For reinforcement learning, we implemented our environment using Gymnasium [Towers et al., 2023]. The environment exposes multiple functions, most importantly, the step() function can be used to take a single step in the environment. A step consists of passing the current observation to the actor, receiving the next action from the actor, computing the reward the action receives and updating the state of the environment.

The actor, critic, and training is implemented using Stable-Baselines3 [Raffin et al., 2021].

The actor receives the current observation and decides on the next action. Since an action in our environment is a token, the actor outputs a probability distri-

bution across all possible tokens. From this distribution, the next token can be chosen deterministically, by choosing the token with the highest probability, or randomly, by sampling the token from this distribution. The randomness of the sampling can be control by a temperature. If the temperature is set to 0, the sampling is equivalent to deterministically selecting the token with the highest corresponding probability. For a very large temperature value, sampling becomes more random, meaning more similar to selection of the next token uniformly at random.

The critic receives the same observation as the actor and outputs an estimate of the reward the actor will receive.

The actor and critic and implemented in a partially shared architecture, as described in Section 5.3.2.

## 5.4 Experiments

### 5.4.1 Fine-tuning Experiments

As a first step, we generated synthetic datasets and fine-tuned BLIP on them. Figure 5.5 shows the accuracy of describing an object or the grid background completely correct. This means that the description for an object contains the correct color, position, object type and shape. Describing the background only requires the background color and the size. The y-axis corresponds to the number of fine-tuning epochs. These accuracies are measured on an evaluation dataset distinct from the training data.

Figure 5.6b shows the accuracy for describing the color, shape, type, and position of an object each individually. Similarly, Figure 5.6a shows the accuracy for describing the background color and the accuracy for describing the size of the grid.

The model quickly learns to describe the grid background, reaching up to 98% accuracy. While the background color seems fairly easy to recognize, learning to predict the correct shape takes somewhat longer.

For the objects, the model quickly learns to recognize color and type. The model learns to distinguish 8 different object types: *pixel, line, diagonal, rectangle, square, cross, diagonal cross* and *random object*. Similarly, ARC tasks also only contain 10 different colors.

Position and shape of an object seems to be harder to learn for our model. Since our grids can have a size up to $30 \times 30$, many more values are possible. However, the model is still able to reach accuracies above 85%.

In Figure 5.7, we show the accuracy for each object type individually. For each object, we consider the accuracy of describing it fully correct, meaning correct

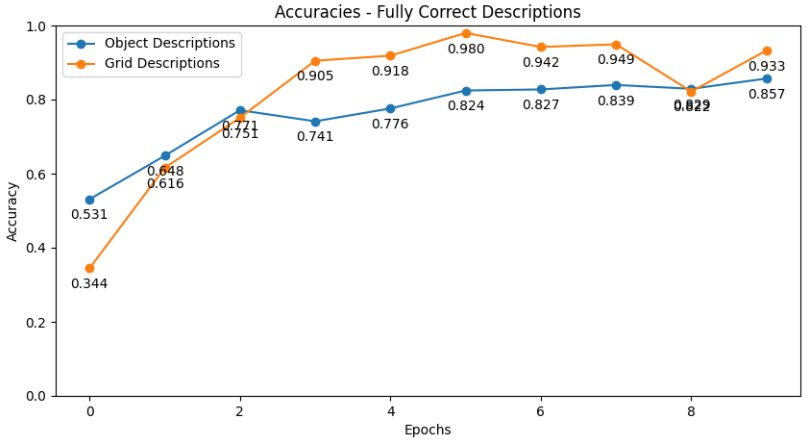



Figure 5.5: Accuracy of describing an object, or the grid background, completely correct.

color, position, shape, and type.

We observe that distinct shapes, especially cross and diagonal cross, seem to be easier to learn. Random shapes seem to pose the biggest problem. They might be harder to identify and correctly describe, since they might only differ in a single pixel from any other object type. Similarly, the position or shape of a random object can depend on a single pixel. Unlike other object where this is the case, lines or diagonals for example, the position of an additional pixel is not fixed.

### 5.4.2 Reinforcement Learning

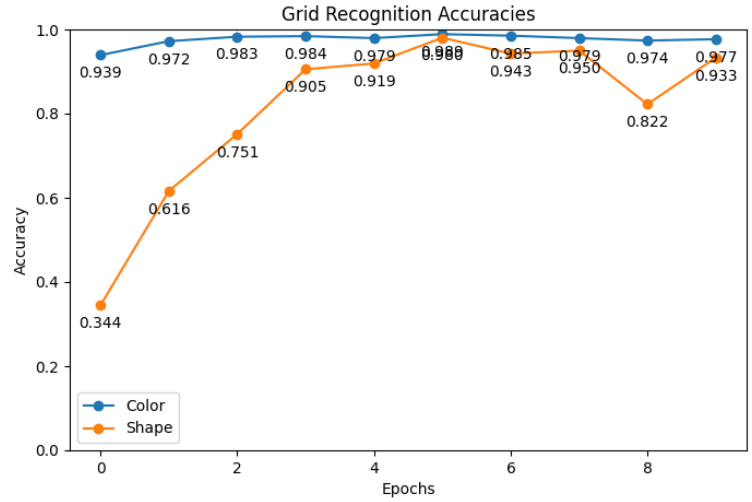
After fine-tuning the captioner, we continued training the same fine-tuned BLIP model using PPO.

In Appendix B.3 in the appendix, we list the parameters we used for PPO training. We also show some additional plots of the reward received during PPO training.

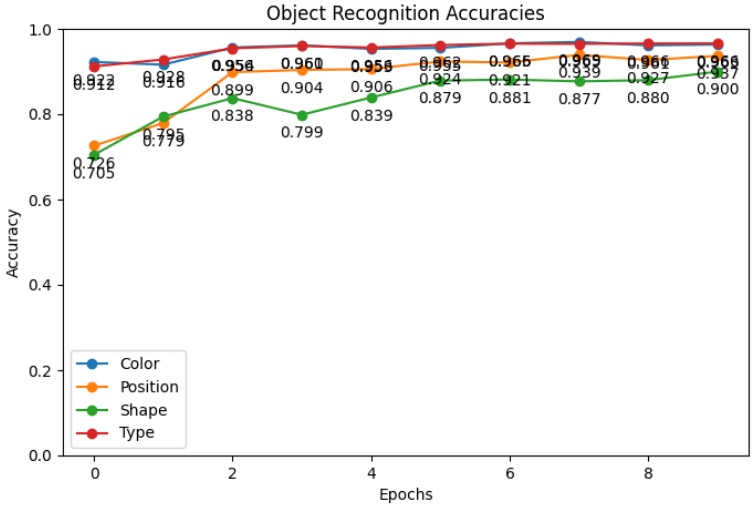
### 5.4.3 Inference

Using the fine-tuned BLIP captioner and the RL-trained BLIP captioner, we again run the full architecture on the ARC training tasks. We used Llama-2 13B as the language model.

In Figure 5.8, we show the results for the fine-tuned BLIP captioner, without PPO training. Our first observation is the small number of solved tasks. We believe that the major cause for this are inaccuracies in the object descriptions.




(a)




(b)

Figure 5.6: (a) Accuracies when describing grid backgrounds. (b) Accuracies when describing objects.

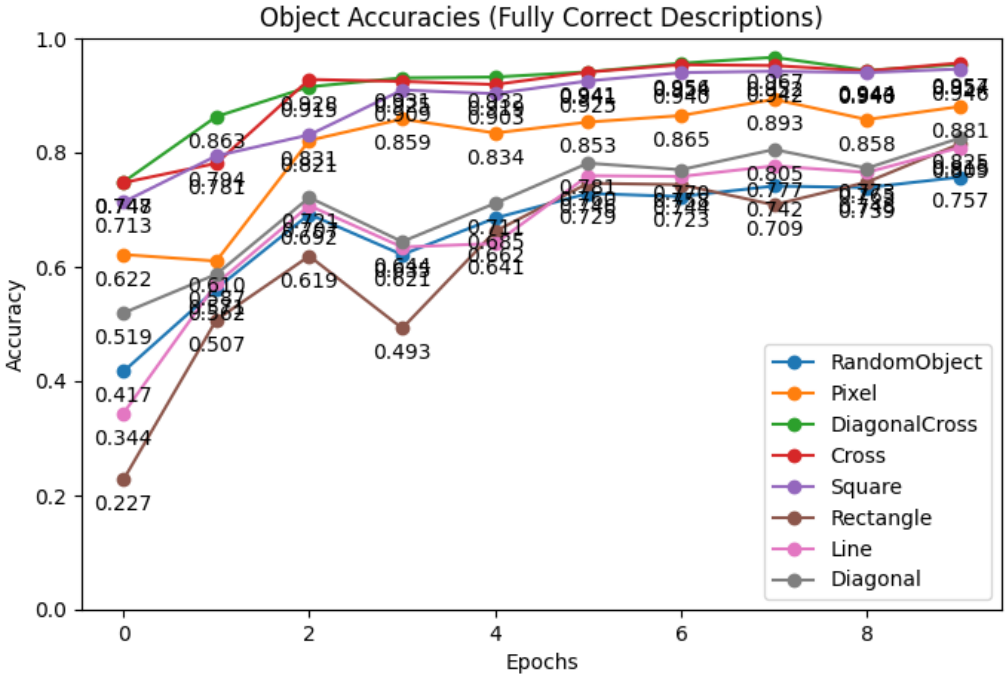



Figure 5.7: Accuracy of describing objects.

Even though the captioner gets most object right in our fine-tuning validation set, it is not fully clear how well this dataset visually matches real ARC grids.

Furthermore, mistakes in object descriptions, for instance a wrong position or shape, can have a negative down-stream effect on the description of the remaining grid. If too many pixels are described for an object, we might mask pixels from another object. This can later cause errors in the description of that object.

If too few pixels of an object are described, some pixels are left and not masked. The captioner might later try to describe those pixels, leading to an unnecessary long prompt and inconsistent object descriptions. This might also partially explain the large number of skipped tasks.

Tasks are skipped if the prompt is too long for the language model. The previously described effect of describing too many objects can be a cause for that. If the captioner produces a wrong caption that leaves some or all pixels of the object unmasked, it will later produce another caption for that object.

On the positive side, we observe a large reduction in failures, from 60 with the hard-coded captioner, to now 21. We find that the hard-coded captioner more frequently describes random objects, on average 5.98 per task prompt. The fine-tuned captioner on the other hand only describes 3.30 random objects on average per task prompt. Since many failures occur due to the language model predicting

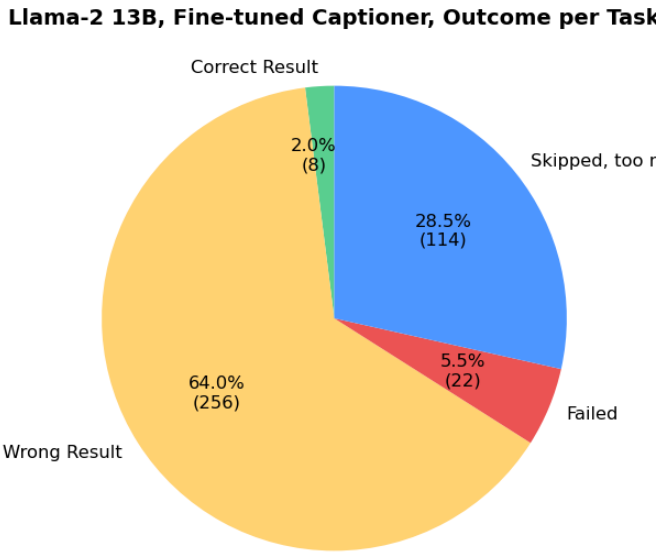



Figure 5.8: Outcome of solving the training tasks using the Llama-2 13B language model with a fine-tuned version of BLIP as the captioner.

wrong random objects in the output, this might be a reason for the decrease in failures.

In Figure 5.9, we show the results for the PPO-trained captioner. We used to fine-tuned BLIP captioner and continued training it using PPO.

From the result, we see that the number of solved tasks did not increase. The biggest change is in the lower number of skipped tasks.

As described previously, errors in object descriptions cause too many or too few pixels masked in the task image. This leads to downstream errors when describing the next objects. During fine-tuning, the BLIP captioner is not exposed to this situation, since previously described objects in our dataset are perfectly masked.

During reinforcement learning, the captioner creates its own training data by interacting with the environment. When the captioner makes a mistake in the description of the shape or position of an object, too many or too few pixels in the mask will be updated. When describing the next objects, the input image is masked accordingly and contains this mistake, exposing the captioner to this issue during training. We speculate that the reinforcement learning taught the captioner to better deal with this challenge.

Unfortunately, since we did not observe a clear improvement in the captioner, it is questionable how much it learned through PPO training.

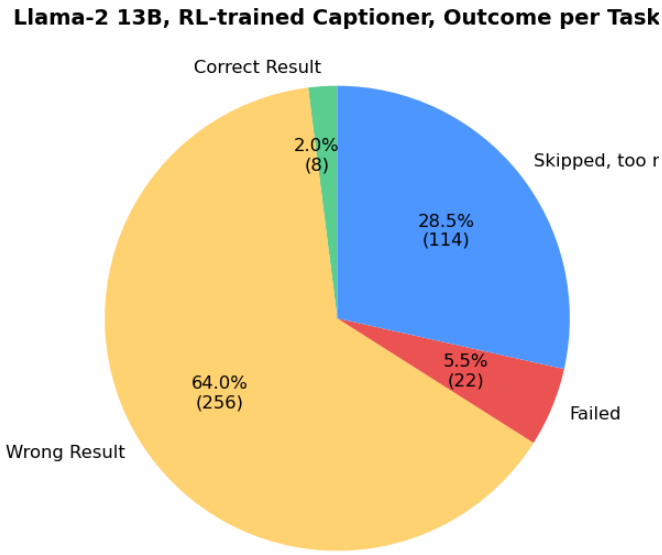



Figure 5.9: Outcome of solving the training tasks using the Llama-2 13B language model with a reinforcement-learning trained version of BLIP as the captioner.

Chapter 6

# Conclusion

We conclude this thesis by providing an overview of our work and our most important insights.

## 6.1 Overview

Our work is based on a previous approach to solving ARC tasks by using language models. This approach turned ARC tasks from images into text by implementing a hard-coded captioner built on hand-craft visual priors.

Using a language model allowed to introduce a large amount of prior knowledge that might be useful to solving ARC tasks. However, language models are purely trained on textual data and might struggle with more visual concepts, like occlusion.

Our objective was to enhance two building blocks of this architecture, the captioner and the language model.

We implemented the language model to be easily replaceable. Numerous language models can be specified as arguments and used for our architecture. We adapted the post-precessing of their output to be much more flexible, and correctly parsing the output of many models. After finding Llama-2 13B to produce good results, we fine-tuned it on the prompts of the training tasks, created by the hard-coded captioner. We found fine-tuning to be most useful in improving the result when only allowing a few guesses.

For the captioner, we aimed to build a learned captioner based on a pre-trained image captioning model. Image captioning model are trained on a large amount of visual data and incorporate a large amount of visual understanding. Ideally, a captioner based on a pre-trained image captioning model can better deal with challenges such as overlapping shapes, handling noise or distinguishing touch objects.

BLIP seemed to be best suited for our application. However, image captioning models like BLI are primarily trained on real-world visual data. We found that

without any additional training, it is not capable of producing useful description of ARC tasks.

To train BLIP for describing ARC tasks, we first fine-tuned it on a synthetic dataset of randomly generated images that resemble ARC tasks. Since we are limited by the length of descriptions BLIP can produce, we decided to describe the objects in an iterative approach. Our fine-tuned BLIP captioner should describe the objects in a grid one-by-one, before describing the size and color of the grid itself. During each of those steps, the captioner receives an image of the current view of the task as input. In that image, we masked out all pixels of objects and grids it previously described. Based on that masking, the captioner infers which object to describe next, or if all objects in a grid are masked out and it describes the background.

For fine-tuning, we implemented a process to randomly generate dataset of image and caption pairs. The images show tasks where some objects and grid may be masked out. The caption is a textual description of the next object the captioner should describe when receiving the corresponding image as input. We found fine-tuning BLIP on that dataset is successful in training it to describe objects and grids with a high accuracy.

Next, we wanted to make use of the training tasks from ARC. Since we do not have any object annotations or captions for these tasks, we cannot directly fine-tune our captioner on them. However, using the captioner and the decoder, we can create descriptions of those tasks, and turn the textual descriptions into pixel values by using the decoder. Comparing the predicted pixel values and the true pixel values can be used for reinforcement learning. We define a reward function based on the number of correct and wrong pixel predictions. Using proximal policy optimization (PPO), we train our model to maximize this reward.

Simultaneously to the training, we implemented the BLIP-based captioner into the architecture. This captioner is initialized by loading any fine-tuned or PPO-trained checkpoint of BLIP. When generating a prompt, it iterates over the task and describes the object (and grid backgrounds) one-by-one.

Finally, we run the full architecture to solve ARC tasks with the fine-tuned BLIP captioner and the PPO-trained BLIP captioner.

In conclusion, our many contributions are:

- We adapted the architecture to allow the quick and easy use of numerous language models. We also implemented fine-tuning on ARC captions for those language models. We showed that different language models vary in their ability to solve ARC tasks, and we found Llama-2 to perform particularly well.
- We built a process to generate large synthetic datasets of random image and caption pairs resembling ARC tasks.

- We implemented fine-tuning and reinforcement learning of BLIP and trained a BLIP-based captioner using those training methods.

## 6.2 Future Work

Lastly, we share some thoughts of future improvement for this architecture or more generally for the use of language and image captioning models for solving ARC-tasks.

- Improving on the dataset for fine-tuning the captioner. The images of the fine-tuning dataset consists of randomly generated objects and grids. The objects are randomly colored, sized and placed. Since fine-tuning had the largest effect on BLIP's ability to describe ARC tasks, building a dataset that more closely resembles real ARC tasks could be beneficial.
- Learning object types. The idea of using a language model is to move away from the hand-crafted rules of DSL-approaches. Similarly, using an image captioning model is motivated by replacing the hand-crafted rules of the hard-coded captioner. However, when generating the fine-tuning dataset, we do so based on hand-crafted objects types (*pixel, line, diagonal, square, rectangle, cross* and *random object*). Learning a set of "base" object types from the ARC tasks could allow the captioner to better segment the task image into objects.
- Building a learned decoder. While we believe that this might not increase the number of solved tasks by a lot, it would still add a lot of flexibility. The vocabulary used by the captioner and language model is currently restricted by what the decoder can understand.
- Further reinforcement learning (RL). In this work, we implemented a RL-approach based on reconstructing the task image from its textual description. Other RL approaches, for example training the full architecture by rewarding correct solutions, might be possible as well.

# Bibliography


Sam Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle Ecanow, Maxwell Nye, Michael Tessler, and Josh Tenenbaum. Communicating natural programs to humans and machines. *Advances in Neural Information Processing Systems*, 35:3731–3743, 2022.

Giacomo Camposampiero, Loïc Houmard, Benjamin Estermann, Joël Mathys, and Roger Wattenhofer. Abstract visual reasoning enabled by language. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2642–2646, 2023.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Raphael Fischer, Matthias Jakobs, Sascha Mücke, and Katharina Morik. Solving abstract reasoning tasks with grammatical evolution. In *LWDA*, pages 6–10, 2020.

Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama, May 2023. URL https://github.com/openlm-research/open_llama.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Aysja Johnson, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. Fast and flexible: Human program induction in abstract reasoning tasks. *arXiv preprint arXiv:2103.05823*, 2021.

Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning*, pages 12888–12900. PMLR, 2022.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

Pankaj Mathur. wizardlm_alpaca_dolly_orca_open_llama_3b: An explain tuned openllama-3b model on custom wizardlm, alpaca, dolly datasets. https://github.com/pankajarm/, https://https://huggingface.co/pankajmathur/, 2023.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

John Chong Min Tan and Mehul Motani. Large language model (llm) as a system of multiple expert agents: An approach to solve the abstraction and reasoning corpus (arc) challenge. *arXiv preprint arXiv:2310.05146*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti

Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023. URL https://zenodo.org/record/8127025.

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.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data. *arXiv preprint arXiv:2309.11235*, 2023.

BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*, 2023.

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199*, 2021.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

Appendix A

# Language Model

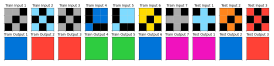
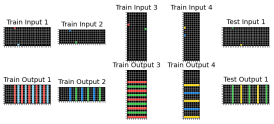
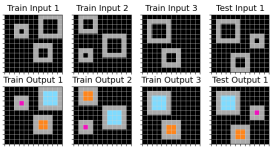
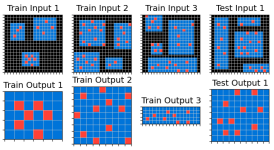
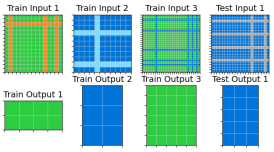
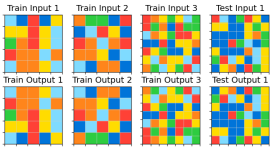
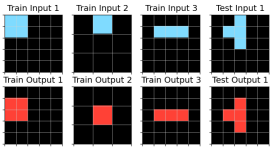
For each language model we tested, Table A.1 shows the number of solved tasks, the number of tasks with a wrong answer predicted, the number of tasks where our architecture failed to produce any answer, and the number of skipped tasks. For each task, the language model made 6 guesses. If at least one guess fully matches the correct answer, a task is considered solved. If our architecture produced at least one answer, but not the correct answer, a task is considered wrong. If the captioner produces a prompt that is too long for the language model, a task may be skipped. If our architecture failed during every guess, the task is considered failed. We used the hard-coded captioner with multichromatic objects and diagonal connections (SMuD) from [Camposampiero et al., 2023].

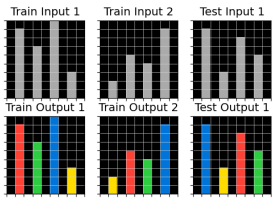
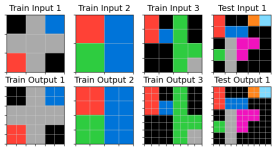
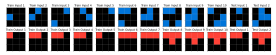
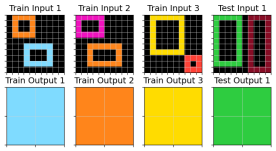
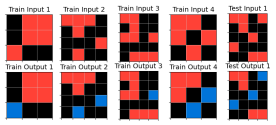
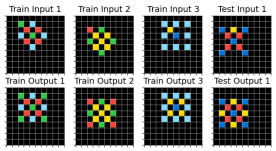
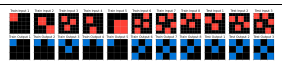
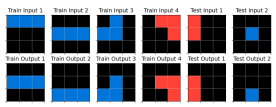
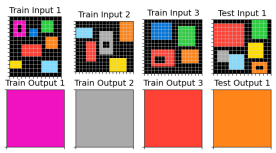
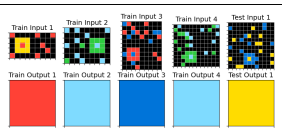
| Language Model | Size | Solved Tasks | Wrong | Failed | Skipped |
|---|---|---|---|---|---|
| Bloom | 560M | 25 | 303 | 60 | 12 |
| | 1.1B | 33 | 295 | 60 | 12 |
| | 1.7B | 32 | 306 | 50 | 12 |
| | 7B | 41 | 311 | 38 | 10 |
| Llama-2 | 7B | 43 | 287 | 54 | 16 |
| | 13B | 50 | 274 | 60 | 16 |
| | 70B | 56 | 257 | 53 | 34 |
| Open Llama | 3B | 45 | 299 | 40 | 16 |
| | 7B | 44 | 300 | 40 | 16 |
| | 13B | 41 | 296 | 47 | 16 |
| Openchat | 13B | 45 | 263 | 76 | 16 |
| OPT | 125M | 0 | 88 | 286 | 26 |
| | 350M | 0 | 62 | 312 | 26 |
| | 2.7B | 21 | 241 | 112 | 26 |
| | 6.7B | 24 | 275 | 75 | 26 |
| | 13B | 26 | 281 | 67 | 26 |
| Orca-mini | 3B | 33 | 291 | 60 | 16 |
| | 7B | 47 | 274 | 63 | 16 |
| | 13B | 50 | 341 | 2 | 7 |
| Vicuna | 7B | 42 | 281 | 61 | 16 |
| | 13B | 47 | 297 | 40 | 16 |
| | 33B | 47 | 284 | 53 | 16 |

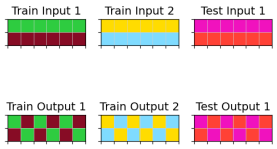
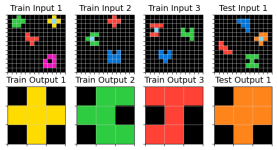
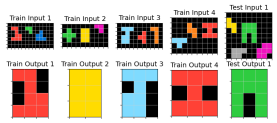
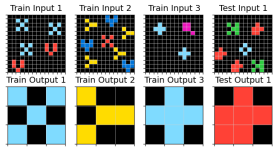
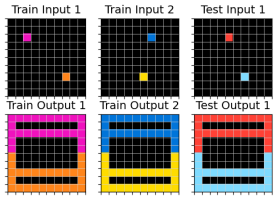
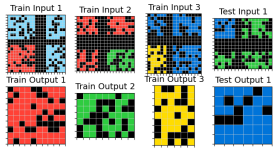
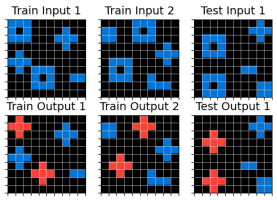
Table A.1: For each language model we tested, this table shows the number of solved tasks, the number of tasks where our architecture predicted a wrong result, the number of tasks where our architecture failed, and the number of skipped tasks.
The model was allowed 6 guesses, and we used the hard-coded SMuD captioner.

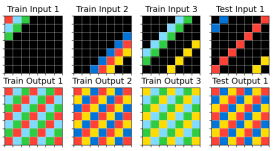
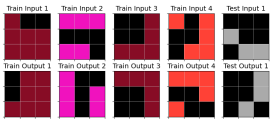
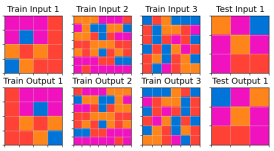
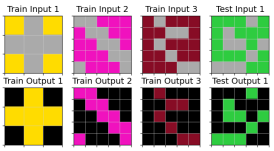
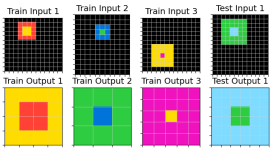
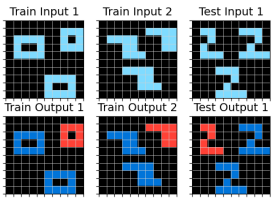
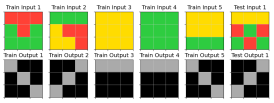
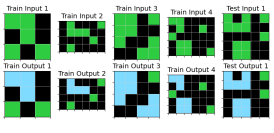
## A.1 Parameter Size

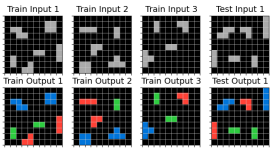
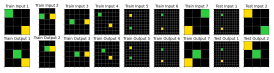
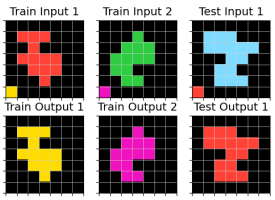
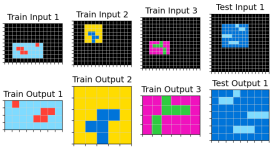
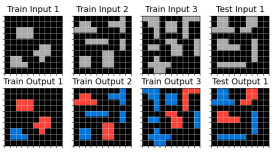
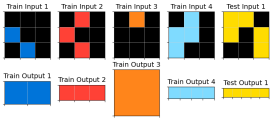
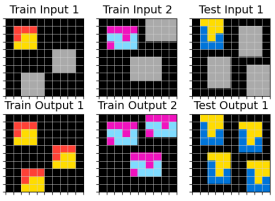
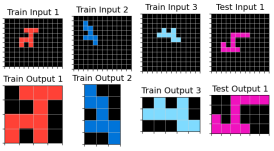
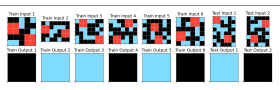
In the following table, we show the training tasks solved for different sizes of the Llama-2 language model. The 3 last columns show which tasks were solved by the Llama-2 7B, 13B and 70B models respectively. If a column contains the ✓symbol, this means that the model solved the corresponding task. The model was allowed 6 guesses, and we used the hard-coded SMuD captioner.

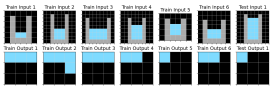
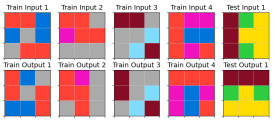
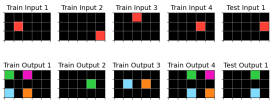
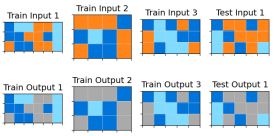
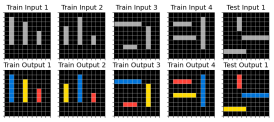
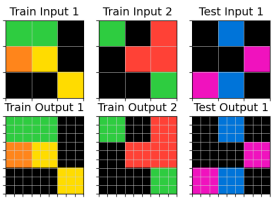
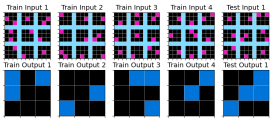
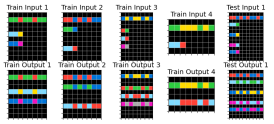
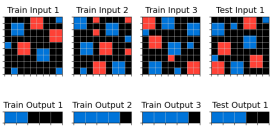
| | Name | 7B | 13B | 70B |
|---|---|---|---|---|
|  | 27a28665 | | | ✓ |
|  | 0a938d79 | | | ✓ |
|  | c0f76784 | ✓ | ✓ | ✓ |
|  | 8efcae92 | ✓ | | ✓ |
|  | 1190e5a7 | | ✓ | |
|  | 68b16354 | ✓ | ✓ | ✓ |
|  | a79310a0 | ✓ | ✓ | ✓ |

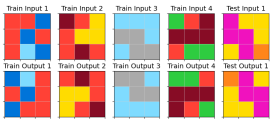
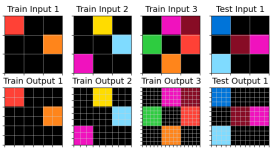
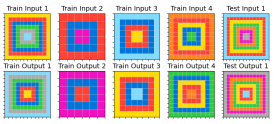
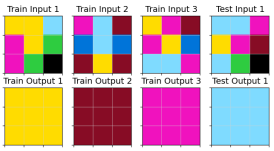
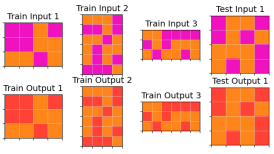
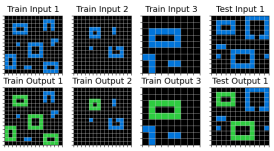
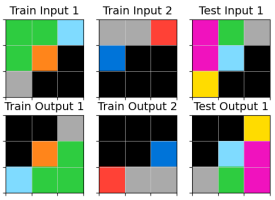
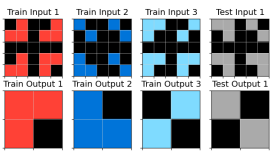
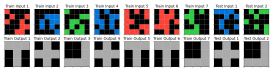
| | | | | |
|---|---|---|---|---|
|  | 08ed6ac7 | ✓ | ✓ | ✓ |
|  | c59eb873 | ✓ | ✓ | ✓ |
|  | 794b24be | ✓ | ✓ | ✓ |
|  | 445eab21 | ✓ | ✓ | ✓ |
|  | aedd82e4 | ✓ | | |
|  | 11852cab | | ✓ | ✓ |
|  | ff28f65a | | | ✓ |
|  | 25ff71a9 | ✓ | ✓ | ✓ |
|  | b9b7f026 | | | ✓ |
|  | d9fac9be | ✓ | ✓ | ✓ |

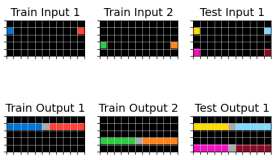
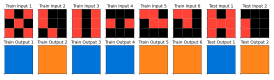
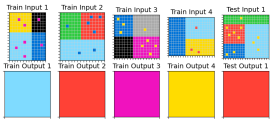
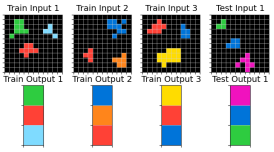
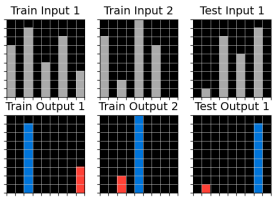
| | | | | |
|---|---|---|---|---|
|  | e9afcf9a | ✓ | ✓ | ✓ |
|  | 5117e062 | | | ✓ |
|  | be94b721 | ✓ | ✓ | ✓ |
|  | 39a8645d | ✓ | ✓ | ✓ |
|  | 1bfc4729 | ✓ | ✓ | ✓ |
|  | 0b148d64 | ✓ | ✓ | ✓ |
|  | 6c434453 | | | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 05269061 | | ✓ | ✓ |
|  | ed36ccf7 | ✓ | ✓ | ✓ |
|  | 67a3c6ac | ✓ | ✓ | ✓ |
|  | f76d97a5 | ✓ | ✓ | ✓ |
|  | b94a9452 | ✓ | ✓ | ✓ |
|  | b230c067 | ✓ | ✓ | ✓ |
|  | 6e02f1e3 | ✓ | ✓ | ✓ |
|  | 67385a82 | ✓ | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 6e82a1ae | | | ✓ |
|  | dc433765 | ✓ | ✓ | ✓ |
|  | aabf363d | ✓ | ✓ | ✓ |
|  | 7468f01a | ✓ | ✓ | ✓ |
|  | d2abd087 | | | ✓ |
|  | d631b094 | | ✓ | |
|  | e76a88a6 | ✓ | ✓ | ✓ |
|  | 1cf80156 | ✓ | ✓ | ✓ |
|  | 239be575 | ✓ | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | b0c4d837 | ✓ | | ✓ |
|  | 74dd1130 | ✓ | ✓ | ✓ |
|  | a9f96cdd | | | ✓ |
|  | c8f0f002 | | ✓ | ✓ |
|  | ea32f347 | ✓ | ✓ | ✓ |
|  | 9172f3a0 | ✓ | ✓ | ✓ |
|  | 6773b310 | | ✓ | |
|  | 82819916 | | ✓ | ✓ |
|  | 1fad071e | | | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 3c9b0459 | ✓ | ✓ | ✓ |
|  | ac0a08a4 | ✓ | ✓ | |
|  | 85c4e7cd | | ✓ | ✓ |
|  | 5582e5ca | ✓ | ✓ | |
|  | b1948b0a | ✓ | ✓ | ✓ |
|  | 810b9b61 | | ✓ | |
|  | 6150a2bd | ✓ | ✓ | |
|  | 88a62173 | ✓ | ✓ | ✓ |
|  | d4469b4b | ✓ | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 29c11459 | | | ✓ |
|  | 44f52bb0 | ✓ | ✓ | ✓ |
|  | de1cd16c | ✓ | ✓ | ✓ |
|  | f8ff0b80 | ✓ | ✓ | ✓ |
|  | a61f2674 | | ✓ | ✓ |

## A.2 Fine-tuning

Table A.3 shows the parameters used for fine-tuning the language model.

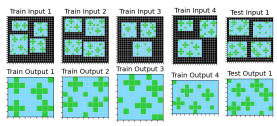
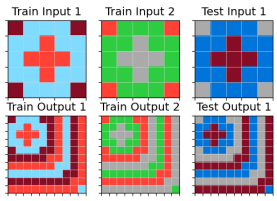
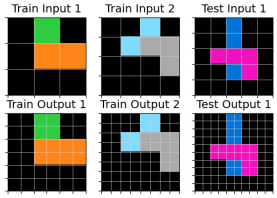
| Parameter | Value |
|---|---|
| LoRA R | 16 |
| LoRA Alpha | 32 |
| LoRA Dropout | 0.05 |
| LoRA Bias | None |
| Steps | 400 |
| Learning Rate | 0.0002 |

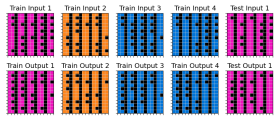
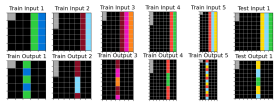
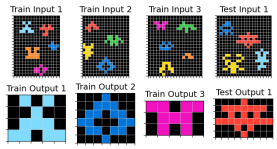
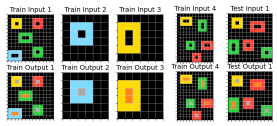
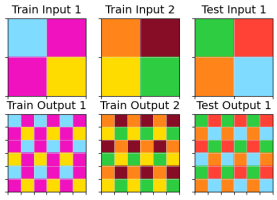
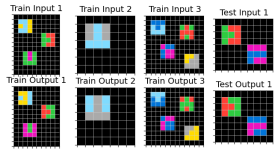
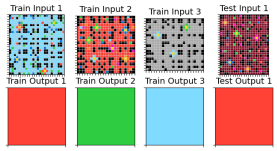
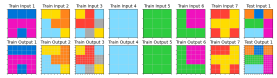
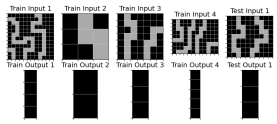
Table A.3: Fine-tuning parameters used for fine-tuning Llama-2 13B.

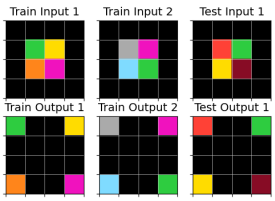
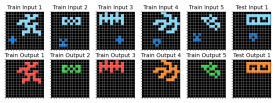
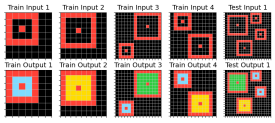
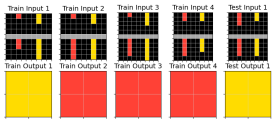
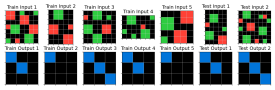
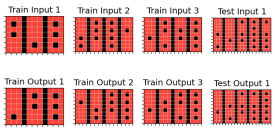
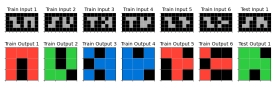
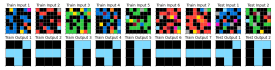
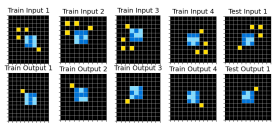
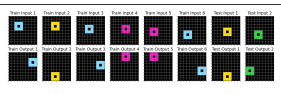
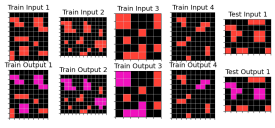
For the first fine-tuning run, we trained on the captions produced from the hard-coded captioner and did not do any reordering. We used all 3 orders: size, left_to_right and random. In total, we trained on 1'200 examples.

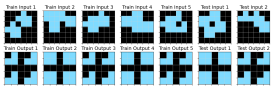
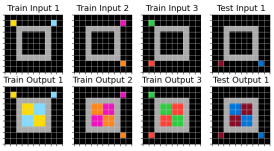
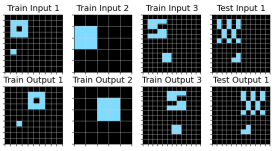
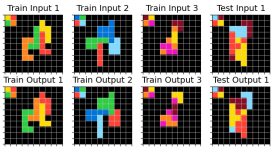
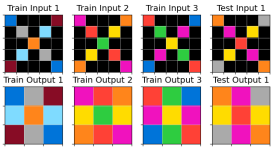
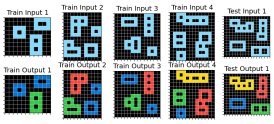
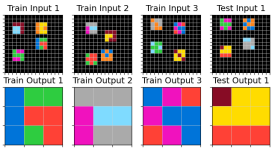
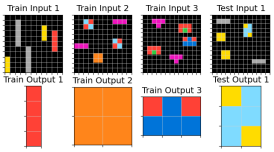
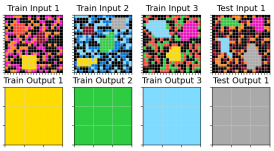
For the second fine-tuning run, we augmented the dataset by reordering the objects in the grid description.

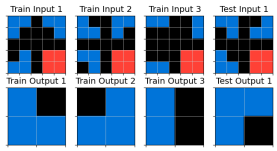
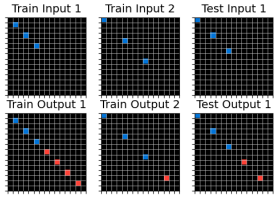
In the following table are the evaluation tasks solved from our fine-tuned language model. Base refers to the Llama-2 13B model without any fine-tuning. FT 1 and FT2 refer to the first and second fine-tuned models. If a column contains the ✓symbol, this means that the model solved the corresponding task.

| | Name | Base | FT 1 | FT 2 |
|---|---|---|---|---|
|  | 2c0b0aff | ✓ | ✓ | ✓ |
|  | 3979b1a8 | ✓ | ✓ | ✓ |
|  | 60c09cac | ✓ | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 85b81ff1 | ✓ | ✓ | ✓ |
|  | e7b06bea | | ✓ | |
|  | d56f2372 | ✓ | ✓ | ✓ |
|  | 84f2aca1 | ✓ | ✓ | ✓ |
|  | 00576224 | ✓ | ✓ | ✓ |
|  | 45737921 | ✓ | ✓ | ✓ |
|  | 642d658d | | ✓ | ✓ |
|  | d4b1c2b1 | | | ✓ |
|  | e872b94a | ✓ | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | 66e6c45b | ✓ | ✓ | ✓ |
|  | 009d5c81 | ✓ | ✓ | ✓ |
|  | 00dbd492 | ✓ | ✓ | ✓ |
|  | 8597cfd7 | ✓ | ✓ | ✓ |
|  | 3b4c2228 | ✓ | | |
|  | 42a15761 | ✓ | ✓ | ✓ |
|  | ed74f2f2 | ✓ | ✓ | ✓ |
|  | 9110e3c5 | ✓ | ✓ | ✓ |
|  | ecaa0ec1 | ✓ | | |
|  | f3e62deb | | ✓ | ✓ |
|  | ae58858e | ✓ | ✓ | |

| | | | | |
|---|---|---|---|---|
|  | b1fc8b8e | ✓ | | ✓ |
|  | 94414823 | ✓ | ✓ | ✓ |
|  | 64a7c07e | ✓ | ✓ | ✓ |
|  | 0becf7df | ✓ | ✓ | |
|  | ca8de6ea | ✓ | | |
|  | 0a2355a6 | ✓ | | |
|  | 9a4bb226 | ✓ | ✓ | |
|  | cd3c21df | ✓ | ✓ | ✓ |
|  | 3194b014 | | ✓ | ✓ |

| | | | | |
|---|---|---|---|---|
|  | be03b35f | ✓ | ✓ | ✓ |
|  | 0b17323b | | ✓ | ✓ |

Appendix B

# Captioner

## B.1 Fine-tuning

| Parameter | Value |
|---|---|
| Epochs | 10 |
| Learning Rate | 6e-06 |
| Batch Size | 16 |
| Frozen Weights | None |
| Base model | Salesforce/blip-image-captioning-base |
| Number of Training Pairs | 272'689 |
| Number of Validation Pairs | 30'489 |

Table B.1: Training parameters used for fine-tuning BLIP.

The following plots show the accuracies of describing the color, position, shape, and type of objects. Each plot shows the result of one object type. The x-axis shows the number of epochs the captioner has been trained for.

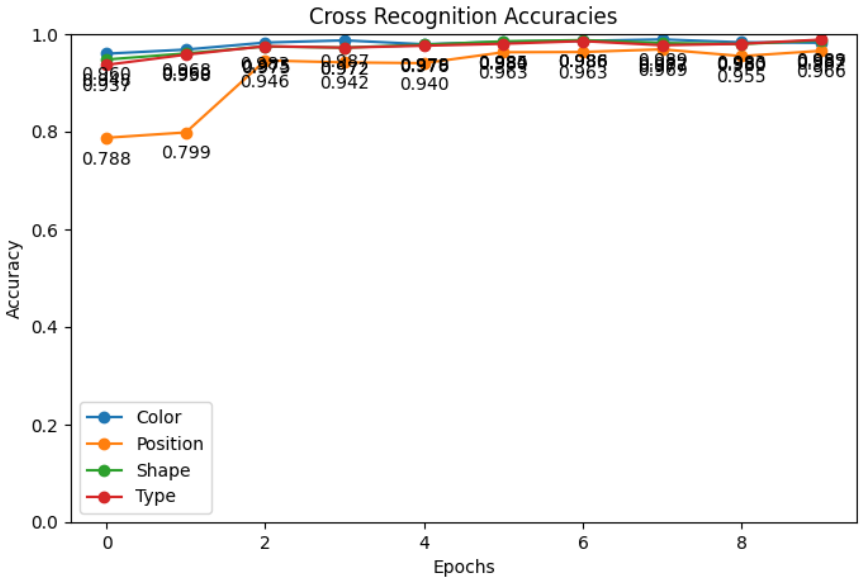



Figure B.1: Accuracies of describing objects of type "cross".

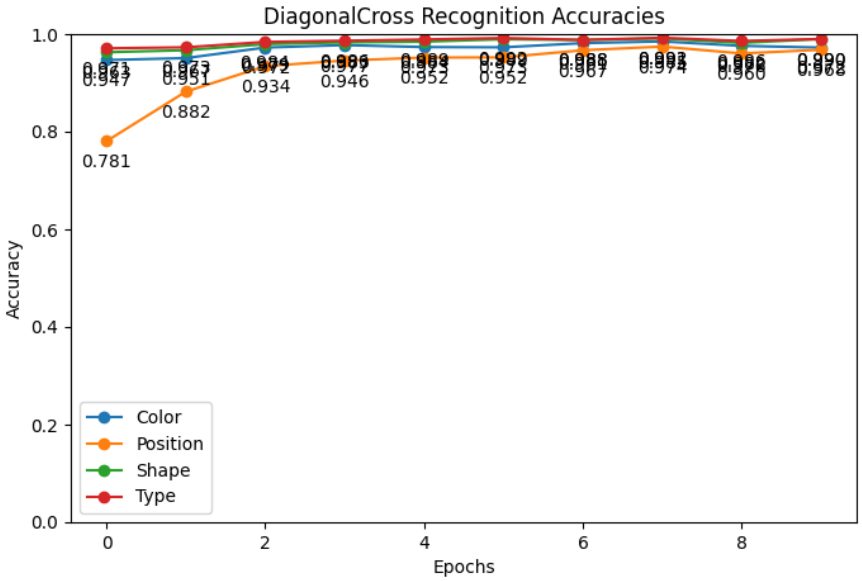



Figure B.2: Accuracies of describing objects of type "diagonal cross".

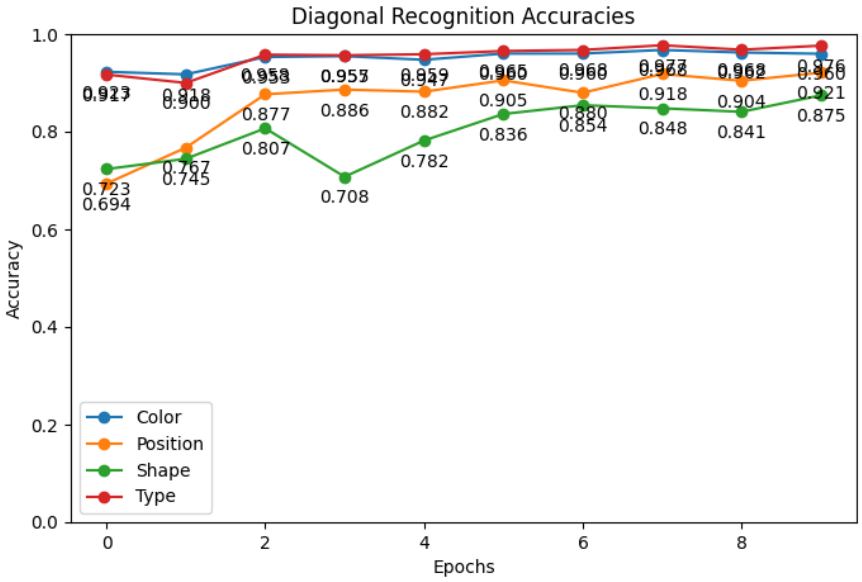



Figure B.3: Accuracies of describing objects of type "diagonal".

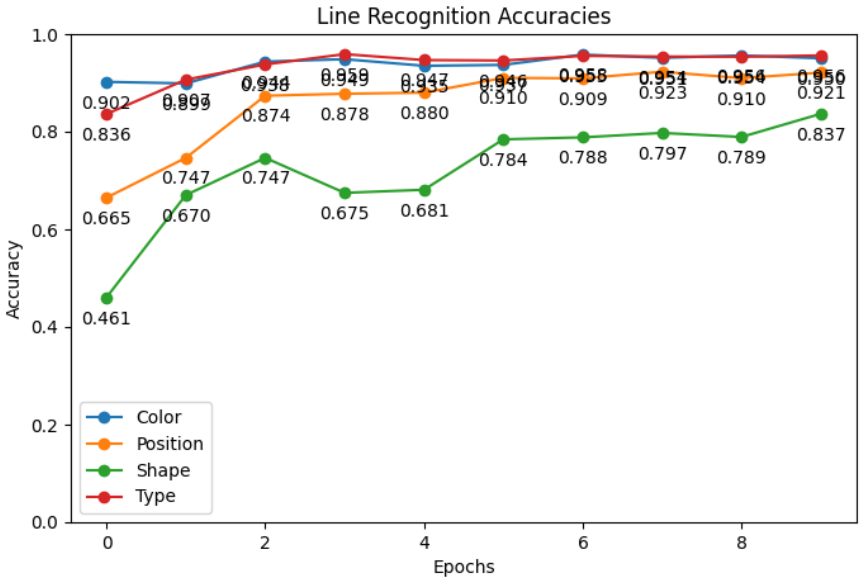



Figure B.4: Accuracies of describing objects of type "line".

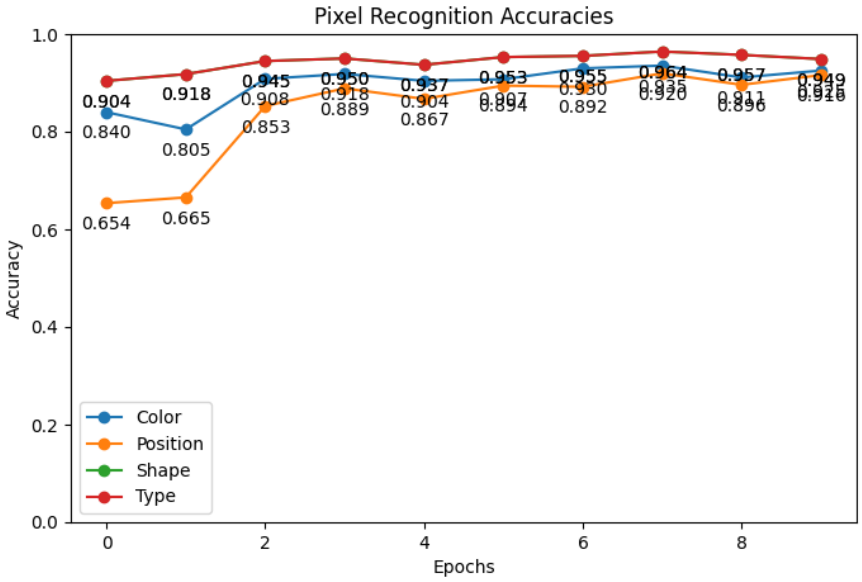



Figure B.5: Accuracies of describing objects of type "pixel".

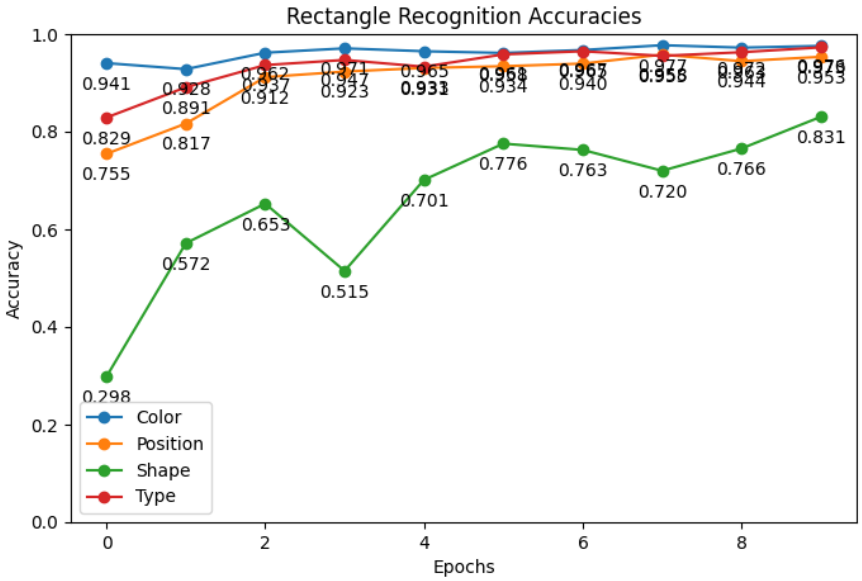



Figure B.6: Accuracies of describing objects of type "rectangle".

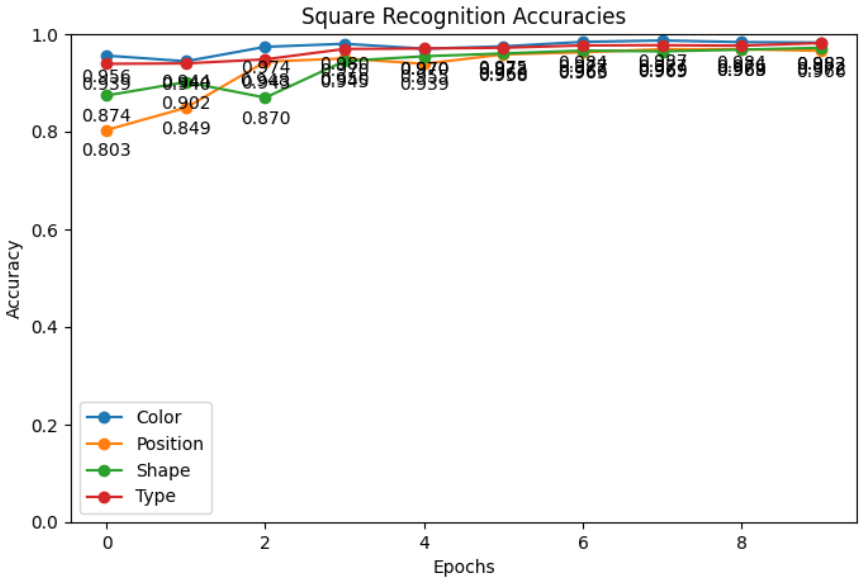



Figure B.7: Accuracies of describing objects of type "square".

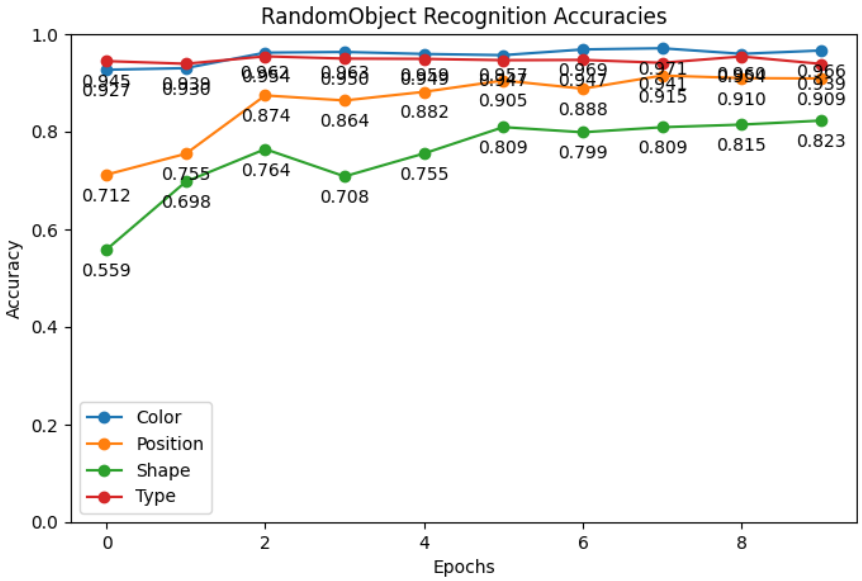
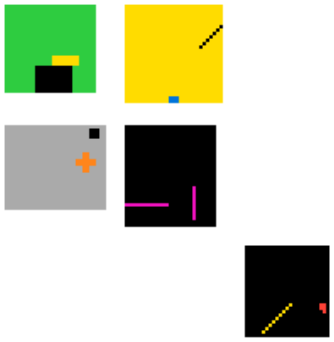
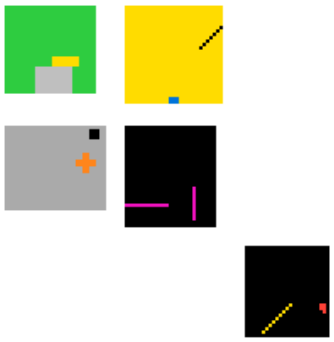
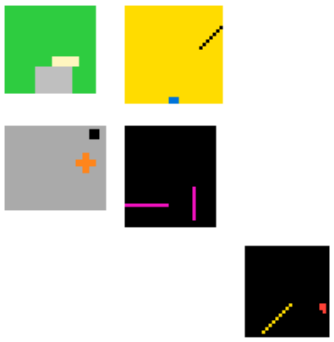
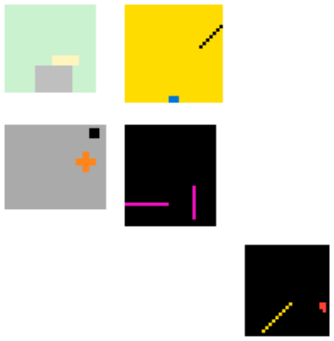
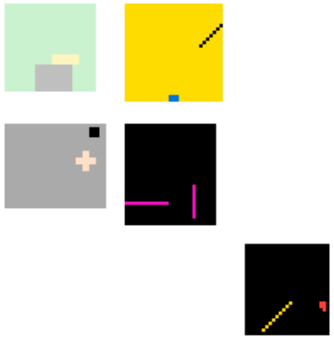


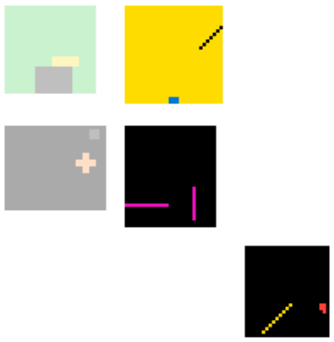
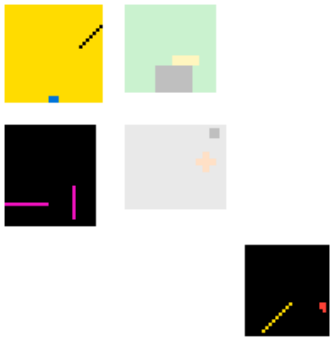
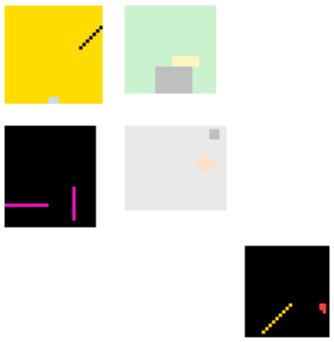

Figure B.8: Accuracies of describing objects of type "random object".

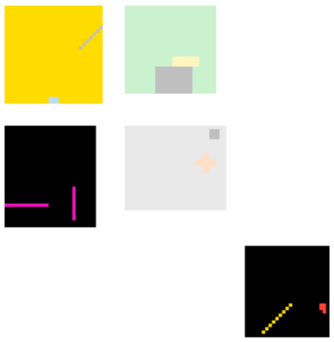
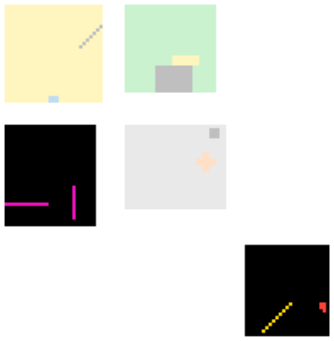
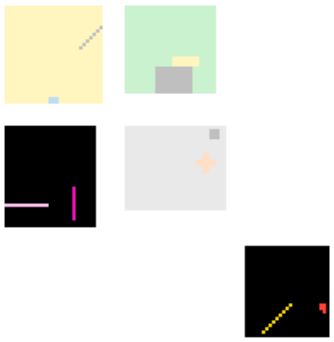
## B.2 Dataset Generation

The following table shows how we iterate over the objects of a synthetic task. Given the image on the left, we train the captioner to produce the caption shown on the right.

| **Image** | **Caption** |
|---|---|
|  | rectangle, with upper left corner in position (18,9), of size 8x11, monochromatic of color black. |
|  | rectangle, with upper left corner in position (15,14), of size 3x8, monochromatic of color yellow. |

| | |
|---|---|
|  | 26x27 grid, green background. |
|  | cross, with upper left corner in position (8,21), of size 6x6, monochromatic of color orange. |
|  | square, with upper left corner in position (1,25), of size 3x3, monochromatic of color black. |

| | |
|---|---|
|  | 25x30 grid, grey background. |
|  | rectangle, with upper left corner in position (27,13), of size 2x3, monochromatic of color blue. |
|  | increasing diagonal, with upper left corner in position (6,22), of length 7, monochromatic of color black. |

| | |
|---|---|
|  | 29x29 grid, yellow background. |
|  | horizontal line, with upper left corner in position (23,0), of length 13, monochromatic of color purple. |
|  | vertical line, with upper left corner in position (18,20), of length 10, monochromatic of color purple. |

## B.3 Reinforcement Learning

| **Parameter** | **Value** |
|---|---|
| Steps per Rollout Collection | 100'000 |
| Total Timesteps | 6'000'000 |
| Learning Rate | 5e-09 |
| ent_coef | 0.01 |
| vf_coef | 0.1 |
| target_kl | None |
| reward function | $c^{2.125}/(c + w^{1.25})$ |
| negative reward per step | $-0.1$ |

Table B.3: Parameters used for PPO-training of the fine-tuned BLIP checkpoint. In the reward function $c$ and $w$ stand for the number of correct pixel predictions and the number of wrong pixel predictions.

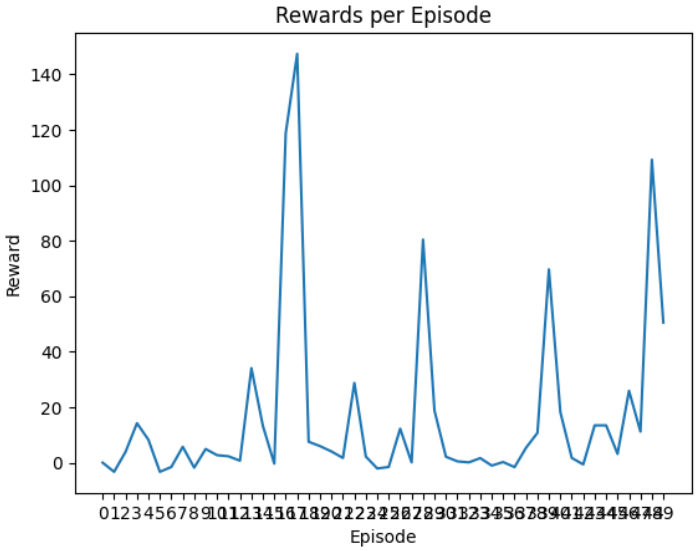



Figure B.9: Reward given during each episode. X-axis show the episode, y-axis the reward value.


Figure B.10: Reward per object or grid description during PPO training. X-axis shows the number of steps, y-axis the reward value.