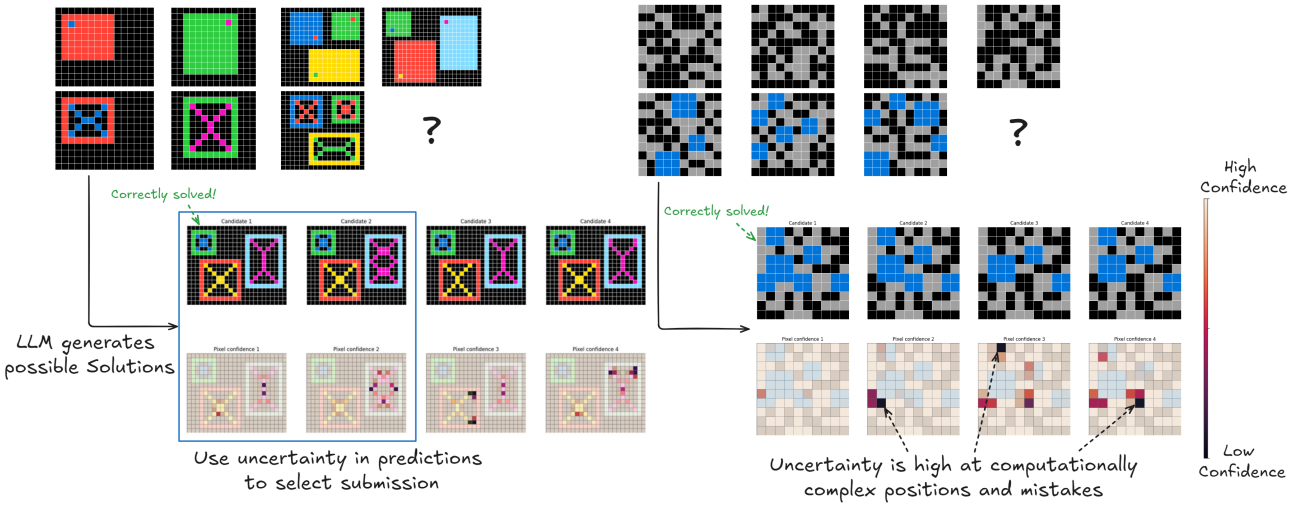
The LLM ARChitect: Solving the ARC Challenge Is a Matter of Perspective

Daniel Franzen*†[1], Jan Disselhoff*†[2], and David Hartmann†[3]

[1]dfranzen.it@gmail.com
[2]JanDissel.it@gmail.com
[3]Lambda, Inc., davidh@lambdal.com

Abstract

Large language models (LLMs) have made impressive progress, but they still struggle with abstract reasoning tasks like the Abstraction and Reasoning Corpus (ARC). Prior approaches have not been able to achieve high scores on ARC, suggesting a gap between current AI systems and human-level reasoning. In this work we approach the problem using tailored data augmentation techniques, optimizing the data format as well as training performance, and leveraging our generative model both as a predictor and as a classifier for good solutions. We are able to maximize performance despite limited computational resources, achieving 53.5 (56.5[1]) points on the hidden test set during the ARC-Challenge 2024. Additionally, we are able to solve 72.5 out of 100 tasks in the public evaluation set while training on the other 300.



*Contributing equally to the Kaggle ARC Prize 2024.
†Contributing equally to this paper.
[1]Run finished after kaggle submission deadline

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across diverse tasks, from natural language processing to code generation. However, the fundamental question of whether these systems can truly "reason" remains contentious in the artificial intelligence community. The Abstraction and Reasoning Corpus (ARC) [1], designed specifically to evaluate genuine intelligence in AI systems, provides a stark illustration of this challenge. Although the challenges appear simple to human test takers, both classical algorithmic approaches [2] and modern neural architectures [3] have struggled to achieve high performance on ARC, painting a bleak picture for artificial reasoning capabilities.
Yet, recent developments suggest this picture might be incomplete. The rapid evolution of language models has produced increasingly capable systems at surprisingly small scales, as demonstrated by models like LLaMA-3.2-3B [4] and Nvidia NeMo-Minitron-8B [5]. Moreover, growing evidence suggests that many perceived limitations of these models may be artifacts of implementation or limitations of language rather than fundamental capability gaps. For example, Allen-Zhu and Li [6] show that models often have awareness of their mistakes, even when unable to correct them. Small mistakes in data modelling can significantly inhibit finetuning performance, not because the problem is too complex, but because a simpler structure might be available for the model to learn [7]. Finally, a growing body of research [8, 9, 10] establishes that apparent failures can frequently be traced to tokenization issues rather than reasoning deficits.
It seems that models often possess the requisite capabilities but struggle to access them effectively, leading to a counter-intuitive conclusion: The challenge for LLMs lies not in the absence of reasoning ability, but in creating conditions that allow these capabilities to emerge.

Building on these insights, we developed an approach specifically tailored to the ARC dataset. Our method solves 72.5 task out of 100 examples of the public evaluation dataset and 56.5 points in a late submission to the Kaggle competition, suggesting that efficient finetuning, proper tokenization and tailored algorithmic support can indeed unlock the latent reasoning capabilities of these systems.

2 Pipeline Overview

Our approach focuses on efficiently fine-tuning a large language model to solve the Abstraction and Reasoning Corpus (ARC) tasks. The approach can be separated into several distinct components: an expanded dataset, secondary fine-tuning at test-time, an inference method optimized to ARC and a heuristic candidate selection algorithm based on the prediction scores.

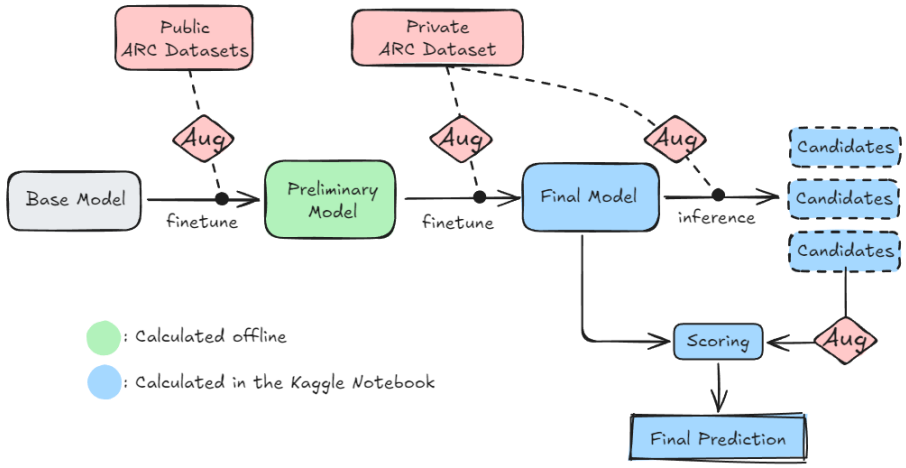


Figure 1: High Level overview of our pipeline. We retrain an LLM on public ARC data, which is then finetuned an additional time on the hidden test cases. Subsequently, this model predicts several solution candidates from which we select two using an algorithmic approach. All blue rectangles are calculated in the 12h timelimit of the kaggle notebook. At several parts in our pipeline we "augment"(Aug) the data by applying transformations to the examples.

We also use data augmentations at train, inference and scoring, substantially increasing our score. The key components of our pipeline are outlined below:

Datasets:
In addition to the official ARC dataset, which is divided into "public train", "public eval", and "hidden test" subsets, we also utilize the Re-ARC dataset by Hodel [11]. This extended dataset is derived from the public training dataset using custom generators written in a domain-specific language (DSL), allowing for the creation of additional, diverse examples. Since the Re-ARC dataset is a superset of the public ARC train dataset, we choose to replace the public train data with Re-ARC.
We additionally use Concept-ARC and ARC-Heavy [12, 13] which we introduce in Section 3.1.

Data Representation:
In order to represent the ARC examples in a denser format, we modify the tokenizer and embedding layers by reducing the number of tokens significantly to only 64 symbols, allowing us to encode the data succinctly and transparently for the LLM. More detailed information regarding our data modeling approach can be found in Section 3.2.

Augmentation:
While data augmentations are commonly used to increase dataset sizes, our approach uses augmentations at every part of the pipeline. Specifically, we

use rotations, transpositions, and permutations of the colors and the order of examples, as these preserve the underlying challenge structure. We use these augmentations not only to increase the amount of training data, but also to increase the variety in our predictions and to provide a way to filter wrong solutions. This contributes significantly to the performance, and will be discussed more extensively in Section 3.3.

Models:
We use the augmented data to fine-tune decoder-only LLMs. Being constrained by Kaggle's 2 Nvidia T4 GPUs, the models have to meet two main requirements: a maximal memory usage of 16GB during training and inference, and a minimum context size of 8000 tokens to handle inference on larger problems. We want to point out two models that worked particularly well in our case: Mistral-NeMo-Minitron-8B-Base [5] and an uncensored version of Llama-3.2-3B-instruct [14].

Preliminary & Secondary Fine-Tuning:
Our models are initially trained on Re-ARC and 75% of the ARC public evaluation dataset, leveraging LoRA adapters [15] as a fine-tuning method. The fine-tuned model is subsequently uploaded to Kaggle, where it undergoes additional fine-tuning on the hidden dataset. For more comprehensive details on training and hyperparameters, please refer to Section 3.4.

Candidate Generation:
After fine-tuning has completed, the final model is used to generate solution candidates. We found that both greedy and multinomial sampling provided suboptimal performance for the challenge. Instead, we designed a custom generator that leverages depth-first search (DFS) to extract all possible completions with a sampling probability exceeding a specified cutoff value. This DFS approach is applied to both the original challenge as well as its "augmented" versions, resulting in a list of candidates for each challenge. This not only improved our score, but also the inference time needed. For an in-depth discussion on inference see Section 3.5.

Candidate Selection:
Finally, given the generated list of candidates, we use the *aggregated log-softmax scores* assigned by the fine-tuned model to select two of them for submission. To make sure that these values are informative we compute them over several augmentations of the challenge. This selection algorithm is highly effective, provided that the correct candidate is among those generated. For a detailed explanation of our candidate selection process, refer to Section 3.6.

Model	Baseline	+ Fine-tune	+ Candidate	+ AugScore	+ DFS
Llama-mix	21.0	35.5	55.5	57.5	63.5
Nemo-mix	**26.0**	**40.5**	**57.5**	**69.0**	**72.5**

Table 1: Performance on the evaluation dataset, adding parts of our pipeline. Baseline score shows performance of our network after preliminary finetuning when taking two samples from generation. Fine-tune adds secondary training on the evaluation dataset. Candidate selection uses augmentation and greedy sampling to generate a candidate for each of 8 different augmentation of the task, using logsoftmax scores for selection. AugScore additionally uses the sum of the logsoftmax probabilites from 8 different augmentation as score. Finally, DFS uses our custom DFS scheme to find all candidates with a sampling probability larger than 10%, at the same time increasing the augmentations during inference from 8 to 16 per task (to reflect the speedups gained by the DFS sampling). Scores were measured on the 100 problems of the evaluation dataset that were not merged into the training set.

3 Methods

Our approach to solving the ARC challenge combines data expansion, multi-stage fine-tuning of language models, and specialized solution evaluation. Below, we explain how these components work together to improve the model's performance while keeping computational costs manageable.

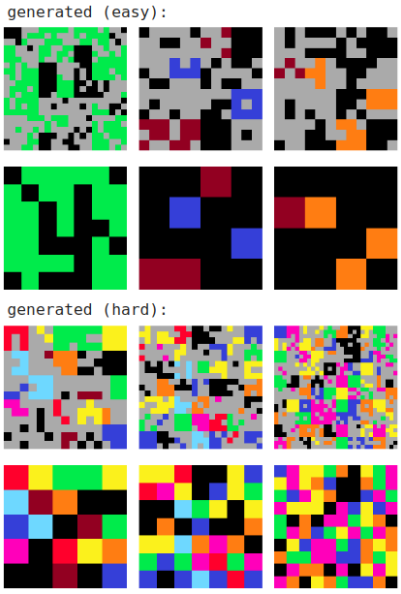
3.1 Datasets

The Abstraction and Reasoning Corpus (ARC) introduced by Chollet [1] challenges the idea that language models cannot effectively generalize from a small number of examples, often referred to as *few-shot prompting*. The original ARC dataset consists of 900 reasoning tasks, divided into 400 training tasks, 400 validation tasks, and 100 hidden test tasks. Each task involves grids of varying sizes, ranging from 1x1 to 30x30, utilizing a palette of ten distinct colors.

Importantly, each task contains only a hand full instances of the respective problem, as illustrated in Figure 2(a). Each instance consists of two grids: one representing the input of the problem and the other representing the expected output. The objective is to infer the underlying mechanics from a few examples and apply this understanding to a new, unseen instance as illustrated in the figure.

Hodel [11] introduced the re-ARC dataset, which re-implements all 400 challenges of the public training dataset. Their code can be used to generate an arbitrary amount of training data for these challenges. An example of code and generated data can be seen in Figure 2.

In addition to ARC and its re-implementation re-ARC, we make use of Concept ARC, which is a second dataset containing similar problems involving




```python
def generate_5614dbcf(diff_lb: float, diff_ub: float) -> dict:
    dim_bounds = (2, 10)
    col_card_bounds = (1, 8)
    noise_card_bounds = (0, 8)
    colopts = remove(5, interval(1, 10, 1))
    noisedindscands = totuple(asindices(canvas(0, (3, 3))))
    d = unifint(diff_lb, diff_ub, dim_bounds)
    cells_card_bounds = (1, d * d)
    go = canvas(0, (d, d))
    inds = totuple(asindices(go))
    numocc = unifint(diff_lb, diff_ub, cells_card_bounds)
    numcol = unifint(diff_lb, diff_ub, col_card_bounds)
    occs = sample(inds, numocc)
    colset = sample(colopts, numcol)
    gi = upscale(go, THREE)
    for occ in inds:
        offset = multiply(3, occ)
        numnoise = unifint(diff_lb, diff_ub, noise_card_bounds)
        noise = sample(noisedindscands, numnoise)
        if occ in occs:
            col = choice(colset)
            go = fill(go, col, initset(occ))
            gi = fill(gi, col, shift(noisedindscands, offset))
        gi = fill(gi, 5, shift(noise, offset))
    return {'input': gi, 'output': go}
```

Figure 2: Examples of Re-ARC challenges and the corresponding code.

"basic spatial and semantic concepts" from the same domain as the original ARC dataset. It contains 160 additional tasks [12].

Our best scoring model also included ARC-Heavy [13], which uses LLMs to generate a large amount of synthetic challenges.

3.2 Data Modeling

In order to apply LLMs to ARC problems, we need to tokenize the data in a manner suitable for our model. This process requires careful consideration of two main challenges:

First, due to the limited context size in typical LLM architectures, an increase of inference time and decline in performance on long context tasks [16], we require a representation that minimizes the number of tokens the model needs to process. Secondly, it is widely recognized that numerous common failure modes in Large Language Models (LLMs) stem from tokenization [8, 9, 10]. For instance, standard tokenization techniques group numbers (some but not all combinations) of one, two or three succeeding digits into dedicated “grouped-digit tokens” [17]. These kinds of merges would complicate ARC tasks unnecessarily.

To address this, we opted to simplify the token set available to the model. In particular, we reduced the number of tokens available from over 120.000 to 64 or less tokens (see Table 2).

Token Category	Tokens	Purpose
Alphabet	`A-Z`, `a-z` (excl. `I,O,i,o`)	Learned pre-prompt tokens
Numbers	`0-9`	Encoding the 10 colors
Newline token	`\n`	Signals end of each grid line
Input/Output	`I, O`	Signals start of problem input/output
Begin token	$\langle bos \rangle$	Inserted once at the beginning
End token	$\langle eos \rangle$	Inserted after each output
Padding token	$\langle pad \rangle$	Internal usage (e.g. batching)

Table 2: Reduced Token Set for ARC-specific LLM Model

Visual Representation of a Challenge Instance:

Compact String Format of same Instance:

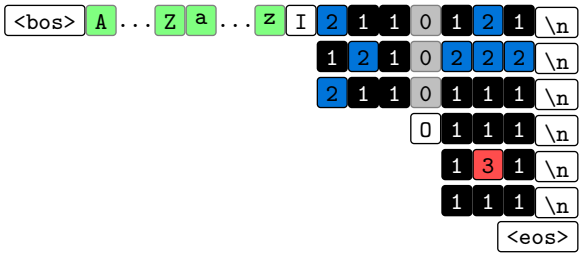


Figure 3: Our standard tokenization approach. Note that we use one token per cell instead of compressing the problem more. We also try to not include any unnecessary delimiters. The Pre-prompt(green) is only included for the first example. Depending on the model and run there might be some small changes to the Pre-prompt and Input/Output tokens.

This reduction offers key benefits. It significantly decreases the model size, as we can remove the majority of rows from the embedding layer. Further, token merges that typically occur during text tokenization are no longer possible. This ensures that the model can focus precisely on the data without the interference of digit separators.
As illustrated in Figure 3, we add a small number of extra tokens to the start of a challenge. Surprisingly, this addition improves the model's performance. We believe that during fine-tuning (where the embedding layers are also trained), the model learns to use these extra tokens as a form of computational buffer, which influences every subsequent token, thereby enhancing overall performance.

We also experimented with adding extra tokens between input and output

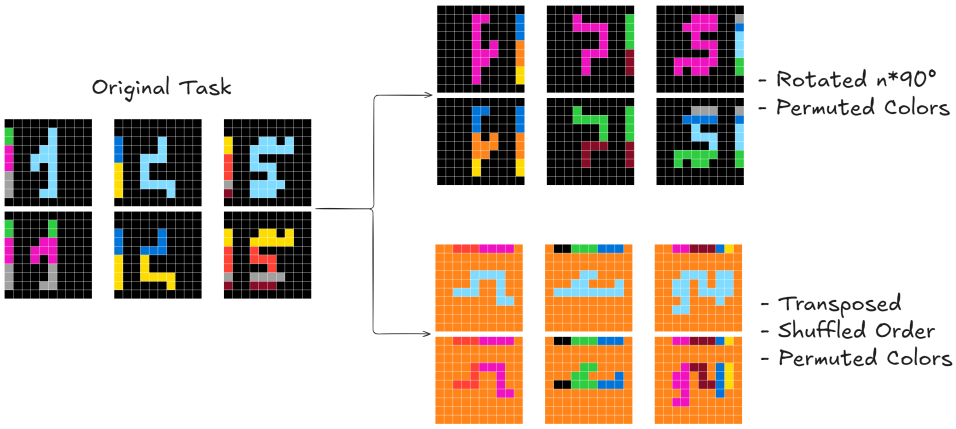


Figure 4: Examples of augmented data. We generally allow rotations and transposition of the examples,as well as permutations of color and shuffling of the order of examples.

of each problem instance. The idea here is to give the model time to internally process the input before having to generate the first token of the output. However, we could not clearly determine if this improved model performance.

3.3 Augmentation

Data augmentation has been a common approach in previous ARC competitions. However, our method extends beyond traditional dataset augmentation, applying transformations throughout our pipeline, both during training and during inference. To formalize our approach, we first need to define the structure of ARC challenges.

A challenge, denoted as $\mathcal{C} = (X_1, Y_1, X_2, Y_2, ..., X^c, Y^c)$, consists input grids X_i paired with their corresponding output grids Y_i. The final pair, X^c and Y^c, represent the challenge grid input X^c and the correct solution Y^c. An augmentation is a function T that maps $\mathcal{C}$ to another challenge $T(\mathcal{C})$ while preserving the challenge's essential structure. While formally defining "essential structure" is difficult as it requires deep understanding of the challenge patterns, we can still identify symmetries of interest.

Our transformations fall into three categories, as illustrated by an example in Figure 4:

- D_8 symmetry operations (rotations and reflections) applied consistently across all grids X_i, Y_i
- Color permutations applied uniformly throughout the challenge, with special consideration for the background color, given by the token 0.

- Reordering of input-output example pairs, which may improve out-of-distribution performance, as evidenced in the literature [7]

While we use the same class of augmentations at each part of the pipeline, the role and requirements of these augmentations vary across our approach: During training, augmentations generate additional valid examples which helps preventing the model from overfitting and provides a broader set of challenges. For inference, we use augmentations to generate a more varied set of solution candidates as outlined in Section 3.5. This requires T to be reversible, enabling us to map the model output to the original un-augmented challenge. In Section 3.6, we use augmentations to evaluate candidates from different perspectives and pick the best two guesses to be submitted. Here the only requirement is that the transformed challenges and solution candidates can be meaningfully evaluated by our model.

3.4 Training the models

Choosing the right large language model (LLM) was essential for achieving strong performance. We tested a variety of different models and found that **Mistral-NeMo-Minitron-8B-Base** [5] performed the best in our experiments. The model is rather large, so efficient fine-tuning methods were necessary to use it effectively.

As a result, we used Low-Rank Adaptation (LoRA)[18], 4-bit quantization and gradient checkpointing, all supported by the unsloth library. We applied the LoRA adaptations to all layers of the network, including the embedding and output layers, for full optimization. We used a learning rate of 1e$-$4 for all layers, except for embedding which had a reduced learning rate of 1e$-$5.

For each challenge $\mathcal{C} = (X_1, Y_1, X_2, Y_2, \ldots, X^c, Y^c)$, we computed gradients only on the outputs Y_i for $i > 1$ and Y^c, so the model never has to predict an input grid, and as it is impossible to correctly predict the first output grid without having seen at least on example. During our second finetuning we compute the gradient on all available outputs.

Preliminary training: The Preliminary training used a LoRA rank of 256 and was done on a single H100 GPU (see Table 3) for various datasets. These included Re-ARC[11], parts of Arc-Eval, ARC-Heavy, and Concept-Arc, with the best-performing configuration using 294.400 examples from Re-Arc, 51.200 from ARC-eval and 22.528 from Concept-Arc. Due to the data augmentations we applied, there were no exact repetitions in the examples. In total, the best performing model was trained on 368.128 examples during this phase.

Secondary training: Secondary training was time-constrained and focused

Dataset	Llama-rearc	Llama-mix	Nemo-mix	Nemo-kaggle
Re-Arc [11]	368 × 400	368 × 400	368 × 400	736 × 400
ARC eval set [1]	-	64 × 300	64 × 300	128 × 400
ConceptARC [12]	-	64 × 176	64 × 176	128 × 176
ARC-Heavy [13]	-	-	-	1 × 200k

Table 3: Number of training examples (epochs × dataset tasks) from each datasets in different training runs.

solely on the hidden test set, using a LoRA rank of 64 and running for 4 epochs. We also tried an additional separate fine-tuning for each task, which increased the score slightly. However, it did not do enough to be considered runtime efficient in our approach and was discarded. Instead, we chose to make use of both $T4$ GPUs available on Kaggle by splitting the test dataset in two parts, and running the full pipeline in parallel on the halves. Using this approach we estimate the training took around 5:20 in the Kaggle notebook.

For a comparison of our training parameters used, see Table 4. In general we found that modern architectures and larger models performed substantially better. In the following sections we will present results for three different models, each trained on different sections of the datasets (see 3). Llama-mix and Nemo-mix are directly comparable models, that showcase the substantial increase in performance from running the $8B$ Nemo model, compared to the $3B$ Llama model. Llama-rearc provides us with a better evaluation set performance baseline, as it has not seen any evaluation examples during training. And finally, the table includes Nemo-kaggle, which was our highest scoring run. However, as its training includes the full evaluation set, we cannot provide any evaluation performance except for our final Kaggle score.

3.5 Solution Inference

Given a trained decoder-only network, standard solution generation is easy. When provided with tokens $x_1, \ldots, x_n$, a model M will calculate the probability distributions $p_{x_2}, \ldots, p_{x_{n+1}}$ for subsequent token predictions. The next token, $p_{x_{n+1}}$, can then be selected either greedily (using argmax) or stochastically (using multinomial sampling). To generate a solution candidate we repeat this procedure until we sample an $\boxed{\langle eos \rangle}$ token, indicating that the example is done. The output is parsed into an array, with some checks to make sure it is a valid grid.

While this can achieve good results, we found both sampling strategies to

	Initial Fine-Tune Locally	Secondary Fine-Tune Locally	Secondary Fine-Tune On Kaggle
Batch size	4	4	2
Gradient acc. steps	2	2	2
LoRA rank	256	64	64
LoRA α	24	16	16
LR (LoRA adapters)	1e−4	1e−4	1e−4
LR (embeddings)	1e−5	1e−5	1e−5
LR schedule	cosine	cosine	cosine
LR warmup phase	25%	25%	100 steps
Weight decay	0.0	0.01	0.01
Model quantization	4 bit	4 bit	4 bit

Table 4: Training parameters for initial as well as secondary fine-tuining. Parameters for Kaggle were chosen differently due to time and memory constraints.

be sub-optimal for generating correct solutions to a given challenge. We attribute these shortcomings to several reasons:

Greedy Sampling: Greedy sampling is straightforward but presents two major issues. First, it offers no guarantees regarding the final sampling probability of the generated solution. It is entirely possible for greedy sampling to produce a solution with very low overall probability: a single high-confidence mistake by the LLM can lead to an undesirable sequence, thereby preventing a successful continuation. Second, greedy sampling inherently lacks variability, yielding only a single result for a given input.

Stochasic Sampling: Standard stochastic sampling techniques introduce more variability but can also repeatedly return the same solution. Moreover, we have no guarantee that the best solution is sampled. If the optimal solution has, for example, a 10% sampling probability, we may need to sample excessively to generate this specific solution. Given that inference is particularly slow – especially for longer tasks – this approach can quickly become computationally prohibitive, even more so within the limitations of a Kaggle notebook.

While there are several alternative sampling schemes, such as sampling with temperature or beam search, we found that a custom sampling algorithm produced the best results for the challenge:

We employ a **depth-first search (DFS)** to explore all possible paths through the solution tree, as long as the path has a cumulative sampling probability greater than a specified threshold p. As soon as a branch falls below this probability, the path is discarded. By leveraging inference caches, this

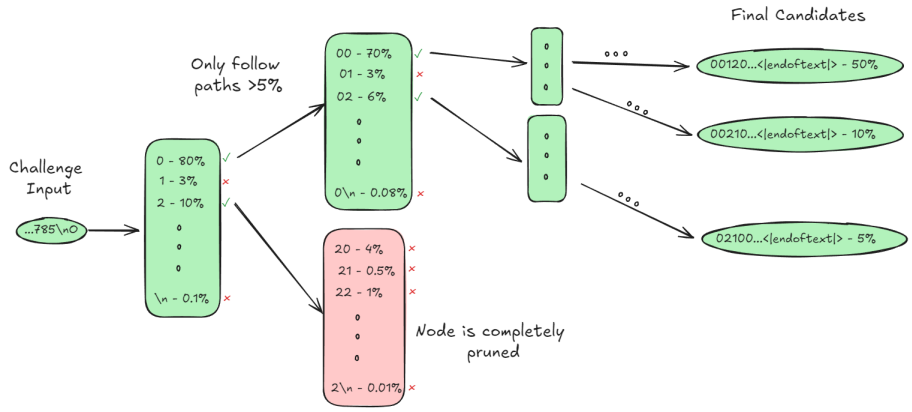


Figure 5: Illustration of the DFS scheme. In this example, only paths with a cumulative probability greater than 5% are kept. Since many paths are cut, we will often end up with far less than 20 possible solutions.

algorithm is able to quickly and efficiently identify *all* potential candidates that have a sampling probability greater than p. This also means that we are guaranteed to extract the solution with the best possible score, provided that it has a probability greater than p – something we cannot guarantee with greedy or multinomial sampling.

We apply this DFS sampling approach across multiple augmented versions of each challenge. The motivation here is that certain solutions are easier for the model to "see" from alternative perspectives. Since we are using a decoder-only text model, it lacks an intrinsic understanding of the 2D structure of these challenges. This limitation becomes evident when solutions that involve critical vertical lines receive a lower probability, while the same challenge, once transposed, is much easier for the model to solve. (see Figure 6)

Consequently, depending on the specific run, we generate DFS candidates across 8 to 16 augmentations for each challenge. This results in a set of candidates that is guaranteed to include the correct solution *if and only if* the model could have sampled that solution with a probability greater than p from at least one augmented perspective.

This strategy is highly effective. Not only do we increase our scores by several points, but the algorithm is also substantially faster then baseline. For a numerical comparison see Table 5. The algorithm is also very memory efficient, as we only need to track a single branch at a time and can avoid duplicating caches – in stark contrast to beam search, which scales linearly with the number of beams considered. We provide pseudocode for DFS sampling in Algorithm 1.

Sampling	Llama-rearc all 400 tasks			Llama-mix 100 tasks			Nemo-mix 100 tasks		
	A	**B**	**R**	**A**	**B**	**R**	**A**	**B**	**R**
Greedy	51.00	60.00	10:51	51.5	65.5	2:36	59.0	75.0	3:49
Stochastic	50.50	59.50	10:58	50.5	65.5	2:39	58.5	72.0	3:55
DFS 17%	51.25	61.50	7:21	51.5	66.5	1:51	60.5	76.5	2:35
DFS 10%	51.00	63.50	9:54	51.5	73.0	2:34	**60.5**	**80.0**	**3:40**

Table 5: Percentage of correctly solved tasks on the 100 randomly split-off tasks of the ARC evaluation set (full evaluation set is used for Llama re-arc) using different sampling strategies on 16 augmented versions (transposition and rotations, as well as randomly permuted colors) of the task. The first column (**A**) denominates the number of tasks correctly solved by the two best-scoring guesses of the model, while the second column (**B**) represents the number of tasks for which the correct solution was present among the candidates obtained during the inference run. Runtimes (column **R**), hh:mm for inference on an Nvidia H100 GPU are given in parentheses. Note that there might be some conceptual leakage from the 300 evaluation challenges used in training of Llama-Mix and Nemo-Mix, possibly inflating their scores a little.

Using this algorithm we are able to generate the correct solution pretty well – in up to 80% of the cases on eval! However, we see a big drop in score when selecting two of these candidates, as we have no way to reliably choose the correct solution from our large number of candidates (yet).

3.6 Selection Strategies

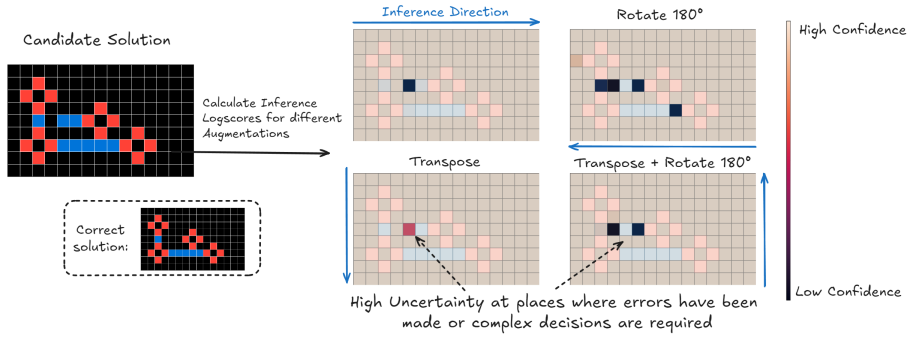


Figure 6: Illustration of the idea behind our candidate scoring. Depending on the augmentation used, the model is able to "see" places of high uncertainty more clear. In this example, the wrong line is barely visible in the transposed version of the task, but when rotating by 180° the model can clearly see that something is wrong. As a result, aggregating these scores provides a highly effective way to filter wrong candidates.

Once we have generated a set of solution candidates, the next step is to determine which ones to submit. Our pipeline up to this point is capable of generating candidates that have a good chance of including the correct solution, but to consider a challenge solved, we must identify it among the

Model	No Aug. Scoring		Scoring with 8 augmentations				
	Best 2 Scores	Similarity Selection	Best Prob	Worst Prob	Sum of Probs	Sum of $\log p$	Time [mm:ss]
Llama-rearc	51.00	51.25	48.38	51.63	48.75	54.5	37:31
Llama-mix	51.5	57.0	57.0	62.0	57.5	63.5	9:50
Nemo-mix	60.5	66.0	66.0	65.5	65.0	72.5	17:05

Table 6: Comparison of different selection strategies using 8 augmentations for each candidate (full evaluation set for Llama-rearc, 100 split-off tasks otherwise). Selecting the candidate with the highest summed log-prob (or alternatively product of probabilities) results in an increase of around 25% over baseline, while only requiring and additional 10-17 minutes on an H100 per 100 tasks. We also compare some alternative aggregation methods. The aggregations compared are: 'best prob' ($max(P)$), worst prob ($min(P)$), sum of probs ($\sum P$) and sum of log prob ($\prod P$) which we use.

candidates, using no more than two guesses.

Recall that any (decoder-only) large language model M also provides us with probabilities for each input token. Given a challenge C and a solution candidate S_k, the model can calculate a probability $P_M(S_k|C)$, which represents how likely it is that the model would generate S_k when provided with C using standard sampling. At first glance, using this probability for selection might seem redundant, as it appears that it would just result in selecting the solution with the highest sampling probability – something that could be achieved more directly by simply sampling.

Instead, we leverage our augmentation techniques. As the network was trained using augmentations, it stands to reason that a correct solution should demonstrate greater stability in its sampling probability under augmented conditions compared to an incorrect one.

We therefore calculate the augmented probabilities:

$$P_i^{aug}(S_k) = P_M(T_i(S_k)|T_i(C))$$

for augmentations $T_1, \ldots, T_n$. Our results always use D_8 symmetry augmentations and optionally color permutations and example shuffling. Using these 8 augmentations for scoring, we observed that taking the product of the probabilities (or alternatively, summing the logsoftmax values) led to a highly stable and effective selection strategy. Our selection strategy can be summarized as:

$$\operatorname*{argmax}_k \prod_i P_i^{aug}(S_k)$$

A comparison of this selection procedure can be seen in Table 6. Adding candidate scoring with augmentations improves our score by roughly 25% over baseline, and shows a significant increase compared to our next best solution. Using this technique we are able to select the correct candidate in 72.5 of 80 possible cases, thereby solving 72.5 out of the 100 randomly split off tasks from the evaluation set.

Even the smaller Llama-rearc model, where preliminary training exclusively used Re-ARC, was able to solve 218 of all 400 evaluation tasks after secondary finetuning on those tasks.

4 Discussion

Using the above pipeline we were able to score 53.5 points during the Kaggle ARC Prize 2024 and 56.5 points shortly after the submission deadline (therefore not being reflected on the leaderboard), securing us the second place in the competition. Our approach demonstrates that LLMs can effectively tackle many of the complex tasks provided by the ARC-challenge. The pipeline presented is highly interconnected and provides results greater than the sum of its parts. The success of our scoring algorithm relies on the custom DFS algorithm's ability to provide candidates with arbitrary cutoff values. This in turn is enabled by optimizing model training in both training phases and by simplifying the problem space. Each part of the pipeline requires the augmentations to work as well as it does.

Finally, we believe that this approach has not yet completely saturated and might scale further, using additional augmentation strategies, further optimizing training hyperparameters or by using larger models.

Acknowledgement

We would like to express our sincere gratitude to Lambda, for providing computational resources essential for optimizing our preliminary training phase within our overall training pipeline. Specifically, they supplied us with a server equipped with 8xH100 GPUs, enabling rapid iteration on our ideas. Their support was instrumental for reaching our final score.

References

[1] François Chollet. "On the Measure of Intelligence". In: *CoRR* abs/1911.01547 (2019). arXiv: 1911.01547. URL: http://arxiv.org/abs/1911.01547.

[2] Icecuber / top-quarks. *Code for 1st place solution to Kaggle's Abstraction and Reasoning Challenge*. Accessed: 2024-11-11. 2024. URL: https://github.com/top-quarks/ARC-solution.

[3] Wenhao Li et al. *Tackling the Abstraction and Reasoning Corpus with Vision Transformers: the Importance of 2D Representation, Positions, and Objects*. 2024. arXiv: 2410.06405 [cs.CV]. URL: https://arxiv.org/abs/2410.06405.

[4] Abhimanyu Dubey et al. "The Llama 3 Herd of Models". In: *CoRR* abs/2407.21783 (2024). DOI: 10.48550/ARXIV.2407.21783. arXiv: 2407.21783. URL: https://doi.org/10.48550/arXiv.2407.21783.

[5] Sharath Turuvekere Sreenivas et al. *LLM Pruning and Distillation in Practice: The Minitron Approach*. 2024. arXiv: 2408.11796 [cs.CL]. URL: https://arxiv.org/abs/2408.11796.

[6] Zeyuan Allen-Zhu and Yuanzhi Li. *Physics of Language Models: Part 3.2, Knowledge Manipulation*. 2024. arXiv: 2309.14402 [cs.CL]. URL: https://arxiv.org/abs/2309.14402.

[7] Zeyuan Allen-Zhu and Yuanzhi Li. "Physics of Language Models: Part 3.1, Knowledge Storage and Extraction". In: *ArXiv e-prints* abs/2309.14316 (Sept. 2023). Full version available at http://arxiv.org/abs/2309.14316.

[8] Aaditya K. Singh and DJ Strouse. *Tokenization counts: the impact of tokenization on arithmetic in frontier LLMs*. 2024. arXiv: 2402.14903 [cs.CL]. URL: https://arxiv.org/abs/2402.14903.

[9] Kaj Bostrom and Greg Durrett. *Byte Pair Encoding is Suboptimal for Language Model Pretraining*. 2020. arXiv: 2004.03720 [cs.CL]. URL: https://arxiv.org/abs/2004.03720.

[10] Kaiser Sun et al. *Tokenization Consistency Matters for Generative Models on Extractive NLP Tasks*. 2023. arXiv: 2212.09912 [cs.CL]. URL: https://arxiv.org/abs/2212.09912.

[11] Michael Hodel. *Addressing the Abstraction and Reasoning Corpus via Procedural Example Generation*. 2024. arXiv: 2404.07353 [cs.LG]. URL: https://arxiv.org/abs/2404.07353.

[12] Arsenii Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. "The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain". In: *Trans. Mach. Learn. Res.* 2023 (2023). URL: https://openreview.net/forum?id=8ykyGbtt2q.

[13] Wen-Ding Li et al. *Combining Induction and Transduction for Abstract Reasoning*. 2024. arXiv: 2411.02272 [cs.LG]. URL: https://arxiv.org/abs/2411.02272.

[14] Chuan Li. *Llama-3.2-3B-Instruct-uncensored*. Accessed: 2024-11-11. 2024. URL: https://huggingface.co/chuanli11/Llama-3.2-3B-Instruct-uncensored.

[15] Yuren Mao et al. "A Survey on LoRA of Large Language Models". In: *CoRR* abs/2407.11046 (2024). DOI: 10.48550/ARXIV.2407.11046. arXiv: 2407.11046. URL: https://doi.org/10.48550/arXiv.2407.11046.

[16] Nelson F. Liu et al. "Lost in the Middle: How Language Models Use Long Contexts". In: *Trans. Assoc. Comput. Linguistics* 12 (2024), pp. 157–173. DOI: 10.1162/TACL_A_00638. URL: https://doi.org/10.1162/tacl%5C_a%5C_00638.

[17] Aaditya K. Singh and DJ Strouse. "Tokenization counts: the impact of tokenization on arithmetic in frontier LLMs". In: *CoRR* abs/2402.14903 (2024). DOI: 10.48550/ARXIV.2402.14903. arXiv: 2402.14903. URL: https://doi.org/10.48550/arXiv.2402.14903.

[18] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

A Miscellaneous

There were several optimizations and ideas that slightly boosted our score but did not fit in the main text.

- We found that we could use the logsoftmax probabilities obtained in the DFS Candidate generation in scoring, providing a slight boost at no additional compute cost.
- Before using scores from various augmentations, we used a selection algorithm based on the similarity of generated outputs, choosing the candidate that had the most pixelwise similarity to all other candidates.
- Due to the nature of the contest we had a very limited time window for generating our solutions. In the end we were able to completely parallelize our solution on both $T4$ GPUs provided.
- There is a trade-off between Generation and Selection, that we were unable to optimize for the hidden test set. Reducing the DFS cutoff value provides more potentially correct solutions, but makes it harder to select them. While the best values for evaluation were rather low, we found higher cutoffs of up to 20% to work better on Kaggle, which had the added benefit of reducing run-time. But even for low percentage values the DFS returns surprisingly few results on average, see Figure 7.
- As can be seen in our results, increasing the size of the base model substantially increases the score at every part of the pipeline. We invested a lot of time to make sure that we are able train and use the $8B$ Nemo Minitron model in a reasonable timeframe on Kaggle. We believe that even larger models could elevate performance again.

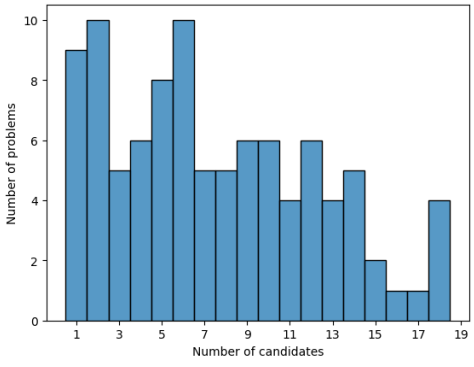


Figure 7: Number of potential candidates returned by the DFS using a 5% cutoff value on the evaluation dataset. In most cases the number of candidates is far lower than the possible theoretical maximum of 20.

Date	Score	Significant Changes	Model	Train [Epochs]	Aug.	Sampling Strategy	Seleciton Strategy
28.08.2024	7.0	Inference only (normal & transpose)	Mistral-Nemo-12B	64	2x	stochastic	-
08.09.2024	20.0	Test-time retraining	Mistral-Nemo-12B	64	2x	stochastic	-
04.10.2024	23.0	Model switch +ReArc + `sim_select`	Llama-3.2-3B (uncensored)	184	4x	stochastic	similarity
06.10.2024	30.0	More inference Aug.	Llama-3.2-3B (uncensored)	184	16x	greedy	similarity
11.10.2024	37.0	Calculating scores w/ augmentations	Llama-3.2-3B (uncensored)	184	16x	greedy	4xAugScore
18.10.2024	41.0	Longer pre-training	Llama-3.2-3B (uncensored)	368	16x	stochastic	4xAugScore
24.10.2024	44.0	DFS inference	Llama-3.2-3B (uncensored)	368	16x	DFS 10%	4xAugScore
05.11.2024	47.0	Model switch	NeMo-Minitron-8B	736	8x	DFS 17%	8xAugScore
06.11.2024	48.5	Longer pre-train + include full eval set	NeMo-Minitron-8B	736	8x	DFS 17%	8xAugScore
09.11.2024	**53.5**	16-fold inference	NeMo-Minitron-8B	736	16x	DFS 14%	8xAugScore
10.11.2024	**56.5***	Adding ARC-Heavy (for pre-train)	NeMo-Minitron-8B	294	16x	DFS 20%	8xAugScore

Table 7: Timeline of scores in the kaggle competition, along with the changes we deem most significant for the improved performance. The final submission marked with (*) finished scoring after the deadline had already ended.

Algorithm 1 Depth-First Probability-Guided Sampling for LLMs

The algorithm presented here assumes that the model supports internal caching for already seen sequences and only needs to process the newly added tokens.

Our actual implementation differs from this simple variant, as we are using *unsloth*, which does not support dynamic caching and requires us to prune the *key-value-cache* of the transformer ourselves.

Furthermore, we use various performance optimizations, like a simultaneous initial forward pass of the best known sequence including prompt and prediction (which is much faster than token-by-token generation) as well as aggregating the sequences during backtracking to avoid the unnecessary processing of sequences that would be discarded later.

procedure DFS_SAMPLE($model, prompt, threshold, max\_len, eos\_id$)
    Input: *model* is the language model
    Input: *prompt* is the prompt that should be completed
    Input: *threshold* is the maximum negative log probability allowed
    Input: *max_len* is the maximum length (including the prompt)
    Input: *eos_id* is the index of the end of sentence token

    function EXPLORE($tokens, score$)
        if $tokens[-1] = eos\_id$ **or** $|tokens| \geq max\_len$ **then**
            return $(score, tokens)$
        end if

        $next\_token\_logits \leftarrow model.predict\_logits(tokens)[-1]$
        $next\_token\_log\_prob \leftarrow -\log\_\text{softmax}(logits)$
        $valid\_sequences \leftarrow \emptyset$

        for each possible next token t **do**
            $next\_score \leftarrow score + next\_token\_log\_prob[t]$
            if $next\_score \leq threshold$ **then**
                $next\_tokens \leftarrow current\_tokens + [t]$
                $continuations \leftarrow$ EXPLORE($next\_tokens, next\_score$)
                $valid\_sequences \leftarrow valid\_sequences \cup continuations$
            end if
        end for
    end function

    return EXPLORE($prompt, 0.0$)
end procedure