Extended Abstract

Motivation Math reasoning is important for language models as the basis for more complex skills that depend on understanding mathematical arguments. But most research occurs with large models, which puts it out of reach for those without access to specialized hardware for training. Because of this, our motivation is to improve math reasoning on a 0.5B parameter model and to demonstrate that it's possible to make the improvements with a consumer GPU using as little memory as possible.

Method First we construct our baseline models. We use SFT on the Asap7772/cog_behav_all_strategies dataset [8] to improve math reasoning and separately use SFT on the Smoltalk dataset [12] to improve instruction following. We also use DPO [11] and RLOO [15] on Countdown [1] to further improve our math reasoning SFT model and use DPO [11] on Ultrafeedback [16] to further improve our instruction following SFT model.

Our approach uses multiple groups of specialized agents to generate several different responses for each prompt and then choose the best response. However, we find that our smaller model struggles with picking the best response as a single task, so we further subdivide the selection process into evaluation and selection and allow the Selector agent to see the Critic agent's outputs. We also add a subtask for extraction that invokes a calculator tool to simplify the inputs and reduce the memory needed for evaluation and selection. Our clear separation of tasks makes it easier to debug the system and for each agent to learn its task. To make the system more robust to mistakes in a single agent, we use several copies of these agent groups, with each agent in each group using a separate LoRA adapter [4] to reduce memory usage. Each agent group outputs its own answer and then they debate by evaluating each other's answers and each selecting the one they think is best. This is repeated a few times until they choose a final response by majority vote.

Implementation The Generator agent uses a base model trained on math reasoning datasets [8] and [1] and outputs 4 different responses. The other agents use a base model trained with SFT on an instruction following dataset [12] because their tasks do not require math reasoning. Their prompts include 2-3 examples of expected input and output to help teach them the task before fine-tuning. The Generator and Selector agents use sampling to output their responses. The Critic agent simply chooses "Correct" or "Incorrect" based on the one with higher logits. The Selector agent chooses the extracted answer with the highest logits conditioned on the prompt.

Results On our evaluation dataset of 200 Countdown-like inputs, our system outperforms each of our baseline models without fine-tuning even with only 1 agent group (SFT improves from 0.316 to 0.458). With 3 agent groups, we see substantial increases compared to 1 agent group (SFT improves from 0.458 to 0.542). When we use automatic extraction instead of the Extractor agent, our system increases its score even more (SFT scores 0.639 with 1 agent group and 0.744 with 3 agent groups). With fine-tuning, our system declines in performance (0.271 with 1 agent group and 0.240 with 3 agent groups), suggesting significant room for improvement in our training implementation.

Discussion Task specialization enabled us to use targeted prompts so each agent could handle its task even before fine-tuning. It also allowed us to quickly debug our system and make targeted changes to the agents making mistakes. Using multiple agent groups improved robustness and using more of them should lead to further reasoning enhancement. Our system without fine-tuning performed better than after our fine-tuning, suggesting that addressing our suboptimal training implementation could lead to substantial improvements. Orchestrating groups of specialized expert agents seems a promising approach for robust and accurate reasoning systems, particularly for small models. We believe this can apply to many different challenging generation tasks.

Conclusion We improve math reasoning in small language models by decomposing reasoning into generation, extraction, evaluation, and selection and by using multiple of these groups with multi-agent debate. With our system, we significantly improve accuracy and robustness even without training. One limitation of our system is that it cannot recover if all generated responses are incorrect, even if it notices this. Future work will focus on the addition of refinement agents that can point out flaws in the reasoning or suggest alternative more unconventional approaches to solving the problem. In particular it will focus on such agents using small language models that run on consumer GPUs.

Improving Reasoning with Multi-Agent Systems

Benjamin Marks
Department of Computer Science
Stanford University
benmarks@stanford.edu

Abstract

Math reasoning is important for language models as the basis for more complex skills that depend on understanding mathematical arguments. But most research occurs with large models, which puts it out of reach for those without access to specialized hardware for training. This work introduces a system with multiple groups of agents that substantially improves math reasoning in small language models. Each agent group works by using one agent to generate several different responses to the prompt, using a second agent to extract each answer and invoke a calculator tool, using a third agent to evaluate for each answer if the input criteria are met, and using a fourth agent to select a correct answer. The agent groups then debate and select a final answer with majority vote. Using multiple agent groups improves robustness by allowing our system to output a correct response even if only 1 of the initial 12 generated responses is correct. Decomposing the task into the subtasks we do makes it much easier to debug our system and quickly improve the agents that are making mistakes. By using Low Rank Adaptation for our fine-tuning, we show a significant reduction in memory compared to similar approaches and are able to train the model on a consumer GPU with only 8 GB of VRAM. We evaluate our system on the Countdown task and show substantial improvement over our baseline models, even with no additional training. We believe this framework of multiple groups of specialized expert agents can be generalized for any challenging reasoning task.

1 Introduction

Math reasoning is an important task for language models because it's a skill that helps unlock several more advanced capabilities and is possible to verify automatically with standardized outputs. But most research on math reasoning occurs with large models that cannot be trained on consumer GPUs. This is especially important for researchers who wish to avoid spending significant compute budgets trying innovative techniques before scaling to larger models.

An example of math reasoning is the Countdown dataset [1], where the model gets a target number and needs to use the other numbers exactly once in a mathematical expression that equals the target. This is challenging because language models are generally unable to evaluate math expressions consistently. Even after fine-tuning on this task, a model may output something invalid. For example, if the target is 76, and the numbers are 32, 36, and 19, it might output something like "Because I can get to 66 with 32+36, I can subtract that from 76 to get 16, which is almost 19." It's also challenging because there are many ways to combine the numbers, and usually very few correct solutions.

In this work, we address these problems with a multi-agent system that can generate several solutions and automatically extract, evaluate, and select the best answers. We use a 0.5B parameter base model and can score 0.543 on a Countdown-like dataset with a base model that barely scores 0.316. We improve upon baseline models trained with standard SFT and RL methods even with no further fine-tuning. Additionally, we optimize for low-memory environments by using memory reduction

techniques that allow training 12 different models concurrently on a single 8GB VRAM consumer GPU.

2 Related Work

2.1 Multi-agent systems

Our initial paradigm of splitting the generation and selection comes from Multiagent Finetuning [2], where Submramaniam et al. describe a multi-agent approach to training a model to self-improve and generate diverse outputs. They first create 2N copies of the model: N generation agents and N critic agents. Then they run M rounds of multi-agent debate, where the first round involves each generation agent generating an output to the input text and subsequent rounds use the critic agents to generate a response based on the summary of the outputs of all agents from the previous round. Multi-agent debate ends with a majority vote for the final output to the initial input. Then they construct the training data for the generation agents by filtering to only the outputs they had that matched the majority vote output. For the critic agents, the training data consist of a weighted sample of the outputs that it successfully corrected to match the majority vote output and the outputs that it successfully did not change and matched the majority vote output. By training each agent only on its own data, each agent learns to specialize and generate diverse reasoning chains.

2.1.1 Task decomposition

Submramaniam et al. [2] use a larger model (4B-8B parameters vs our 0.5B parameter model), and smaller models such as ours can struggle to learn how to critique a response effectively. In "Generator evaluator-selector net for panoptic image segmentation and splitting unfamiliar objects into parts" [3] Eppel and Aspuru-Guzik describe how generating a full solution is much harder to do consistently. While their work focuses on the computer vision task of panoptic segmentation, the same applies to language modeling, and they include language modeling examples in their arguments. Similar to Submraniam et al. [2], they separate generation and evaluation, but they further subdivide the evaluation task into evaluation and selection. They also apply the evaluator and selector independently to separate parts of the generated image, which gives more feedback to the generator and allows them to combine the parts together. They show that even with a model that initially has low quality but high variability, they are able to use it in their system to generate good results when they have high quality evaluator and selector models.

2.2 Reducing memory usage

Submramaniam et al. [2] point out that training and inference are much more expensive and take much longer due to having multiple copies of the model. In our approach, we use Low Rank Adaptation (LoRA) [4] to significantly reduce the memory costs. LoRA works by freezing the initial model and instead adding a pair of trainable matrices of weights to each Transformer layer for rank decomposition. It significantly reduces the number of parameters and memory required for training a model but does not increase the inference time or reduce model quality.

2.3 Tool usage

In Toolformer [5], Schick et al. describe how models can teach themselves to use tools in a self-supervised manner with only a few human-annotated examples. They accomplish this by first converting their existing dataset to a new dataset that has API calls for tools and then using the new dataset to fine-tune an LLM to inject API calls to these tools when the input context suggests a tool might be helpful. Constructing the dataset with tool usage is done by prompting the model with in-context learning examples to suggest possible API calls for each input text in the training data, executing those API calls, and filtering the results to only the calls where both the input and output of the API call help the model better predict the rest of the input text.

As Schick et al. [5] point out, models with fewer than 775M parameters struggle to learn how to effectively use tools. By vastly simplifying the tool-calling paradigm, we are able to allow a model that was trained only on generic instruction fine-tuning to consistently invoke a tool with only 2 in-context examples and no further fine-tuning.

2.4 In-context learning

In "LIn "Language Models are Few-Shot Learners" [6], Brown et al. show that language models can show performance improvements at inference time without fine-tuning by having the models learn from the context within their prompt. They show that adding several examples to the prompt of what the model should do can help it figure out what the task is and how to accomplish it. While fine-tuning on a sufficient amount of data will show better performance than in-context learning, in-context learning is often a good approach for models that have yet to be trained and where training data is limited.

2.5 Chain of Thought

In "Large Language Models are Zero-Shot Reasoners" [7], Kojima et al. show how prompting a model to show its work can help improve its performance on reasoning tasks at the cost of using more compute at inference time. They try out several different prompting techniques and find that "Let's think step by step" has the best performance. They show that this works even in the Zero-Shot setting, where the model has not been fine-tuned on examples with this prompt and where there are no examples in the prompt itself.

3 Method

3.1 Baseline approaches

As described below, our system uses both math reasoning and instruction following to generate answers and select the best one, so we first fine-tune base models to use. For math reasoning, we use SFT on the Asap7772/cog_behav_all_strategies dataset [8] to teach the model how to use Chain of Thought reasoning and for instruction use SFT on the Smoltalk dataset [12]. We also use Direct Preference Optimization (DPO) [11] and Reinforce Leave-One-Out (RLOO) [15] on Countdown [1] to further improve our math reasoning SFT model, and we use DPO [11] on Ultrafeedback [16] to further improve our instruction following SFT model.

3.2 Multi-agent system

As described by Eppel and Aspuru-Guzik [3], it is much harder for a model to consistently generate a correct output than it is to evaluate and select the best output. Initially we attempted to follow Subramaniam et al.'s [2] approach to use a single agent that can evaluate the initially generated responses, but it was difficult coming up with a prompt that would allow our small model to do that effectively and it was hard to determine whether the model was incorrectly evaluating a response as correct or whether it was selecting a response that it had already determined was incorrect. We discovered Eppel and Aspuru-Guzik's [3] approach of splitting the evaluator into separate models for evaluation and selection, and we find it works much better, with our critic model scoring accurately classifying 90% of responses even without training. We also add an extractor agent that can extract the relevant part of the generated response to use in the evaluation and selection inputs.

Each agent is a separate LoRA [4] fine-tune that has its own loss function, and the loss for each agent is based on what its output should be if its input were correct. This helps each agent train independently and be able to improve even if other agents are low quality.

We also find that separating the agents for extraction, evaluation, and selection makes it significantly easier to debug the multi-agent system to determine which component is failing and what we can do to improve it. For example, if the selector agent chooses an incorrect example, but the critic agent has classified it as correct, that means that the critic agent needs to improve, and the selector agent's loss could still be low.

3.2.1 Agent group overview

First, the Generator agent generates several different responses to the prompt. Next, the Extractor agent extracts the answers from each response. Then, the Critic agent evaluates each answer for correctness. Finally, the Selector agent chooses the final answer for the agent group to output.

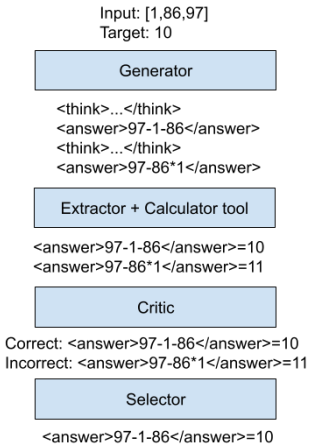

Figure 1: An example of an input being processed by a multi-agent group in our system. First the Generator agent takes the input and generates several different responses. Next, the Extractor agent extracts the answers from each response and invokes the calculator tool to compute the expression value. Then, the Critic agent evaluates each answer and prepends if it is "Correct" or "Incorrect". Finally, the Selector agent chooses one of the answers as the group's final response.

3.2.2 Generator agent

The Generator agent's role is to generate several different possible answers, each with an expression that satisfies the criteria of the prompt. It constructs a prompt instructing it to generate a response that outputs an expression that equals the target and that uses the other numbers in the input exactly once. It has the "User" give it those instructions and then starts its response with "Assistant: Let me solve this step by step." See Appendix: Prompts for the full prompt. This Chain of Thought "Let me solve this step by step" comes from Kojima et al.'s [7] best performing prompt in their paper and helps the model explain its thoughts on its way to a final answer. The prompt instructs the model to output the answer between <answer></answer> tags, and this helps it be easier to extract later.

3.2.3 Extractor agent

The Extractor agent's role is to generate a response that extracts the answer expression from each sampled response from the Generator agent. We add this agent to the Generator-Evaluator-Selector framework from Eppel and Aspuru-Guzik [3] because the Critic and Selector agents do not need to reason over the entire response, and the extra chain-of-thought context in the response confuses them.

The Extractor agent is the only agent that is allowed to use the calculator tool, which it invokes to compute the value of the expression in the answer. It invokes the calculator tool by generating a token with $=$ or $\approx$ in it, and then the calculator tool parses the previous tokens to determine the expression and then uses the Python eval function to compute the value.

Unlike Schick et al.'s [5] Toolformer, we do not need to fine-tune our model with examples of calculator usage because our tool calling API (i.e. checking for $=$) is much simpler. However, we do include several instructions in our prompt for how to invoke the calculator tool and 2 different examples of extracting a response and invoking the tool with $=$.

Because our Extractor agent is often the worst performing of our agents, we also include a variant of our system that replaces the Extractor agent with code that programmatically extracts the answer from the generated response and uses the calculator tool on that.

3.2.4 Critic agent

The Critic agent's role is to evaluate the answer with the computed value to determine if it satisfies the criteria of the problem. For Countdown [1], that means that the expression must evaluate to the target value and that each other number in the input must occur exactly once in the expression. Because the Extractor agent already used the calculator to include the expression value, the Critic agent's job

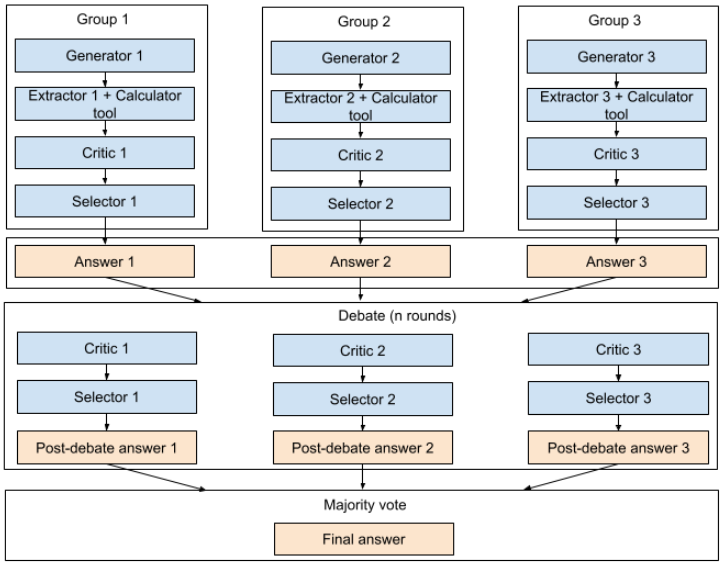

Figure 2: Our system architecture with multiple agent groups. Each agent group processes the input independently as described above. In each round of multi-agent debate, the Critic agent in each agent group evaluates all agent groups' responses for correctness, and each Selector agent selects one of them to output. After several rounds of debate, the system's final answer is selected by majority vote.

is much easier and does not require it to use math reasoning to compute the expression value. The Critic agent's prompt instructs it to output either "Correct" or "Incorrect" and includes 1 example of an "Incorrect" input and 1 example of a "Correct" input.

Unlike the Generator and Extractor agents which use sampling to generate their responses, the Critic agent runs its prompt through the model and predicts logits for the next token. If the logits for "Correct" are higher than the logits for "Incorrect", it outputs "Correct" and vice-versa. This is much more efficient than sampling a token and trying to parse it.

3.2.5 Selector agent

The Selector agent's role is to choose the best answer to the problem. Because the Critic agent has already determined which answers are correct, the Selector agent's job is much easier since it just needs to pick any answer that is annotated as "Correct". The input to the Selector agent consists of the list of answers extracted by the Extractor agent, each prefixed by either "Correct" or "Incorrect" as determined by the Critic agent (see Figure 1 for an example). To simplify the task, we instruct the model to output the first "Correct" answer, even though any of the "Correct" responses would be valid. The Selector agent is instructed to select "None of the above" if none of the answers are correct. In our prompt, we include 2 examples with a selected answer and 1 example where it selects "None of the above".

Similar to the Critic agent, the Selector agent runs its prompt through the model and does not sample a response. However, the Selector agent constructs N+1 different versions of its prompt, each ending with the agent selecting a different answer - one for each of the extracted answers and one for the "None of the above" answer. Instead of sampling tokens to generate the selection, the agent computes log probabilities for all the answers and chooses the answer with the highest mean log probability. This is more efficient than sampling and is much more likely to have the agent select a valid response instead of generating a new (likely incorrect) answer.

3.3 Agent groups

Above we describe how the different agents work together to generate and select a good response. However, due to the sequential nature of the task decomposition, a poorly performing agent can lead to incorrect responses from the whole system. Subramaniam et al. [2] use multiple copies of their

generator and critic agents and use majority vote to select the most common answer. This reduces variance because if one agent in one of the groups performs poorly on a prompt, it is still possible for the other groups to generate a good response and have the majority vote choose that. We use the same paradigm in our system and have 3 copies of each agent, using majority vote to select the most common answer among the groups.

Another technique we adapt from Subramaniam et al. [2] is to use multi-agent debate. In their approach, each critic agent sees the entire generated response from each generator agent before it outputs its own response. We use a simpler approach, primarily to reduce memory usage and generation time. In our approach, we have each agent group output its Selector agent's response. And then we have each agent group process all agent groups' responses with their Critic and Selector agents to produce a new answer. We repeat this up to 3 times and then use majority vote. If all agent groups output the same answer, we end early and use that as the final output of the system. The multi-agent debate technique helps when only 1 agent group outputs a correct answer, and the other agent groups can recognize that correct answer and output it too, improving the robustness of the system.

Since each agent is trained only from its own group's data, they become specialized and start having diverse responses relative to each other.

4 Experimental Setup

4.1 Data

We use the Countdown dataset [1] for training our model and we evaluate on a sample of 200 inputs that are similar in style to it. We use the prompt from the Asap7772/cog_behav_all_strategies dataset [8] to tokenize the input. We use the two stage (format and correctness) evaluation approach from TinyZero [9] to score the outputs. Correct answers get a score of 1, incorrect answers in the right format get a score of 0.1, and answers in the wrong format get a score of 0. For training, we first split the dataset into a consistent 90% train and 10% validation split and use the validation split to gauge when to stop training. For training data used in constructing the baseline models see the Appendix.

4.2 Base models

All base models start originally from the Qwen2.5 0.5B parameter base model [10].

For the Generator agent, we use 2 different base models. Both are SFT trained on the Asap7772/cog_behav_all_strategies dataset [8] to learn Chain of Thought reasoning strategies. One of them is further fine-tuned with Direct Preference Optimization (DPO) [11] on the Countdown dataset, which significantly improves its ability to consistently generate correct responses.

For the Extractor, Critic, and Selector agents, we use a model that we SFT trained on the Smoltalk dataset [12] to improve their ability to follow the instructions in the prompt. None of these 3 agents need any math reasoning to complete their task. This SFT training significantly improves their ability to output the correct responses even without further fine-tuning.

4.3 Hyperparameters

In our experiments, we have the Generator agent output 4 samples to memory and time constraints. We use a relatively low temperature of 0.5 during sampling to help the model be more likely to output the tokens with higher probability in its response. The Generator agent is allowed to output up to 512 new tokens during training and up to 1024 during evaluation. The Extractor agent was allowed up to 40. We use the AdamW [13] optimizer with a learning rate of 1e-5. Due to GPU memory limitations, we use a mini-batch size of 1 and accumulate 8 mini-batches before each gradient step. We allow up to 3 debate rounds per prompt. For hyperparameters used in constructing the baseline models see the Appendix.

4.4 Hardware

Experiments are run on our consumer desktop. It has an Nvidia RTX 2070 Super GPU with 8GB of VRAM, and a Ryzen Threadripper 1900X CPU with 32 GB of RAM.

4.5 Experiments

We compare the following models and systems: our SFT math reasoning model, our DPO math reasoning model, our RLOO math reasoning model, and our multi-agent system. For our multi-agent system, we compare the performance with 1 agent group vs 3, with automatic extraction vs the Extractor agent, and with fine-tuning vs no fine-tuning. For experiments comparing different baseline models used within our multi-agent system see the Appendix.

5 Results

Model	Number of agent groups	Extractor	Score	Score - at least one correct generation	% at least one correct generation
SFT	N/A	N/A	0.316	N/A	N/A
DPO	N/A	N/A	0.748	N/A	N/A
RLOO	N/A	N/A	0.73	N/A	N/A
Untrained agent system	1	Agent	0.458	0.674	63.5%
Untrained agent system	3	Agent	0.542	0.663	79.0%
Untrained agent system	1	Automatic	0.639	0.950	63.5%
Untrained agent system	3	Automatic	0.744	0.915	79.0%
Fine-tuned agent system	1	Agent	0.271	0.363	68.5%
Fine-tuned agent system	3	Agent	0.240	0.281	80.0%
Fine-tuned agent system	1	Automatic	0.604	0.926	61.0%
Fine-tuned agent system	3	Automatic	0.712	0.875	79.0%

Table 1: We compare a few configurations of our system with the baselines. Agent extraction refers to using the Extractor agent with the calculator, while automatic replaces that with code that extracts and computes the value. For our system, we show the score on all prompts as well as the score on the prompts where the Generator agent successfully outputs at least one correct response.

5.1 Quantitative Evaluation

Table 1 shows a comparison of our results on the 200 inputs in our evaluation set across several versions of our system and the baselines. Our multi-agent system improves the results of the baseline SFT model it uses for the Generator agent from 0.316 to 0.458 with a single agent group and 0.542 with 3 agent groups without any fine-tuning. When we use automatic extraction, it increases to 0.639 and 0.744 respectively, which is on par with the baseline RL methods. The baseline RL methods each took over 24 hours of training, while our system requires no training. Due to bugs in our training implementation, our system regresses during fine-tuning, especially with the Extractor agent, scoring 0.271 with 1 agent group and 0.240 with 3 groups.

We also measure the score when the Generator agent successfully outputs at least one correct response to show that our system is quite good at selecting the best generated response, scoring 0.915 with automatic extraction and 0.663 with the Extractor agent.

5.2 Qualitative Analysis

5.2.1 Benefits of task specialization and agent groups

Task specialization allowed us to use more focused prompts that allow each agent to do its task well, even before any fine-tuning. We were able to debug our system with metrics indicating how well each agent is doing and make targeted adjustments to the prompts of agents that were making mistakes. The modularity allowed us to try out different components, different base models, different prompts, and even do an ablation on replacing the Extractor agent with code.

Using several agent groups with multi-agent debate improves robustness, helping the system output a correct answer even if only 1 out of the initial 12 responses is correct. Using more agent groups will likely lead to further improvements. Allowing the Selector agent to select "None of the above" is

particularly useful with multiple agent groups because it helps the agent groups pick another group's answer instead of an arbitrary incorrect answer from the first agent group.

5.2.2 Error analysis

Our fine-tuned system performs worse than the untrained system. This is especially surprising for the automatic extraction version since the Selector's accuracy increased during training and the Critic's accuracy was consistently 98-100%. This clearly indicates our training implementation is suboptimal, and future work should focus on updating this (e.g. with different learning rates, different loss functions, etc.).

The system is quite reliant on the Extractor agent to both successfully extract the answer and to invoke the calculator to compute the expression. Especially for less consistent generation models (i.e. SFT base), the overall quality degrades significantly with the Extractor agent vs the automatic extraction.

The Critic agent is quite good at classifying answers that evaluate to a number that isn't the target as "Incorrect", but it is much worse at classifying answers that fail to use each input number exactly once. In practice, this is less of a concern because most of the time such answers also do not evaluate to the target number and are thus able to be classified appropriately.

The Selector agent frequently fails to select "None of the above" even when the Critic agent marks all the answers as "Incorrect". In the single agent group system, this is actually a positive because the incorrect answer is scored as 0.1, while "None of the above" would score 0. But with multiple agent groups, this is unhelpful because of the ability to pick another group's answer.

6 Discussion

6.1 Implementation challenges

Our system is sensitive to the prompts we use for each agent, and it took several tries for us to find ones that allow the agents to complete their tasks. This was particularly challenging for the Extractor agent, which needs several repetitions of how to invoke the calculator tool. Even with that, it still frequently generates other text instead. We also frequently exceeded our GPU's memory, so we had to reduce our generation token limits during training to fit on the GPU. During fine-tuning, we see the accuracy of the agents either remaining constant or increasing, but still see a significant decrease in score after training.

6.2 Broader implications

Our system with more granular agents improves upon Subramaniam et al.'s [2] multi-agent system by making the tasks simpler so smaller models can successfully complete them. It also means that agents can be trained with LoRA [4] instead of a full fine-tune, resulting in significant memory reductions. It is also faster since the Critic and Selector agents (as well as multi-agent debate) do not require sampling responses and thus take <1 second each.

By generalizing Eppel and Aspuru-Guzik's [3] Generator-Evaluator-Selector to language models and putting it in multiple agent groups like Subramaniam et al. [2], we also create an architectural pattern that we believe can apply to many challenging generation tasks. Building a robust reasoning system involves orchestrating groups of several small specialized expert agents instead of a large model that tries to do everything well on its first try. The multiple groups makes it robust to errors, while the specialization within each group makes it easy to debug and to compose several agents together. It also allows running them on consumer hardware with smaller GPUs or even CPUs.

7 Conclusion

We address the problem of improving math reasoning in small language models by focusing on the brittleness of consistently generating a single correct response and augmenting a math reasoning model with a multi-agent system that can generate many responses and pick the best ones. We demonstrate that by decomposing reasoning into generation, extraction, evaluation, and selection and

by using multiple of these groups with multi-agent debate, we can significantly improve accuracy and robustness even without fine-tuning, as shown by our results on the Countdown dataset.

7.1 Future work

One limitation of our system is that there is currently no mechanism by which the agent system can recover if none of the generated responses are correct, even if the system recognizes that this is the case. One simple approach is to use the Selector agent's "None of the above" output as a signal to retry generation. But this may not be helpful because the Selector agent often does not select that even when the Critic agent has indicated none of the answers are correct. A more promising and general approach would be to consider different refinement agents that can point out flaws in the reasoning or can suggest alternative more unconventional approaches to solving the problem. This is something that DeepMind proposed for science reasoning with "Towards an AI co-scientist" [14], but for a model 3 orders of magnitude larger. We believe that future work should investigate how to use similar approaches for refinement in smaller models that can run on consumer GPUs.

8 Team Contributions

- **Benjamin Marks:**
  - Data loading for Asap7772/cog_behav_all_strategies, Smoltalk, Countdown, and UltraFeedback
  - SFT, DPO, RLOO implementations
  - Generator, Extractor, Critic, and Selector agents and their loss functions
  - Calculator tool invocation and implementation
  - Multi-agent debate, majority vote selection
  - Evaluation scripts and notebooks
  - LoRA finetuning, Weights & Biases integration, HF hub integration

Changes from Proposal Due to unforeseen circumstances, the external person working on this project was not able to contribute.

We focused our efforts primarily on the math reasoning task instead of both math reasoning and instruction following. We did not run RLOO on the UltraFeedback dataset, and we did not implement tools for instruction following. Additionally, our tool invocation strategy was not as extensive as we described in the proposal (e.g. we did not construct a fine-tuned dataset to make it aware of tool calls, and we did not extend the tool calling to support multiple tool calls). We did not apply per-agent restrictions on the use of tools and instead restricted the usage to only a single class of agent (the Extractor agent).

Instead of having more groups of agents like Submraniam et al. [2], we instead used 2 additional agents per agent group (Extractor and Selector) to simplify the tasks for smaller models. We also made significant optimizations to focus on memory usage, allowing our system to train efficiently on GPUs with less than 8GB of VRAM.

References

[1] Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (sos): Learning to search in language, 2024. URL https://arxiv.org/abs/2404.03683.

[2] Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains, 2025. URL https://arxiv.org/abs/2501.05707.

[3] Sagi Eppel and Alan Aspuru-Guzik. Generator evaluator-selector net for panoptic image segmentation and splitting unfamiliar objects into parts, 2020. URL https://arxiv.org/abs/1908.09108.

[4] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, 2021. URL https://arxiv.org/abs/2106.09685.

[5] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, ... Dario Amodei. Language Models are Few-Shot Learners, 2023. URL https://arxiv.org/abs/2005.14165.

[7] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners, 2023. URL https://arxiv.org/abs/2205.11916.

[8] Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs, 2025. URL https://arxiv.org/abs/2503.01307.

[9] Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. Tinyzero. https://github.com/Jiayi-Pan/TinyZero, 2025. Accessed: 2025-05-26.

[10] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, ... Zihan Qiu. Qwen2.5 Technical Report, 2025. URL https://arxiv.org/abs/2412.15115.

[11] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. arXiv preprint arXiv:2305.18290, 2023.

[12] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025. URL https://arxiv.org/abs/2502.02737.

[13] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization, 2017. URL https://arxiv.org/abs/1711.05101.

[14] Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, Khaled Saab, Dan Popovici, Jacob Blum, Fan Zhang, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Pushmeet Kohli, Yossi Matias, ... Vivek Natarajan. Towards an AI co-scientist, 2025. URL https://arxiv.org/abs/2502.18864.

[15] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms, 2024. URL https://arxiv.org/abs/2402.14740.

[16] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Ultrafeedback: Boosting language models with scaled ai feedback, 2024. URL https://arxiv.org/abs/2310.01377.

A Additional Experiments

A.1 Constructing baseline models

This section describes how we construct the baseline SFT, DPO, and RLOO models we use for comparisons.

For all training runs we accumulate 16 minibatches of size 1 for each gradient step and use the AdamW optimizer with a learning rate of 1e-5. For DPO for both math reasoning and instruction

following, we use $\beta = 0.1$ to encourage exploration away from the reference model. For DPO and RLOO on math reasoning, the base model is the SFT model for math reasoning, and for DPO on instruction following, the base model is the SFT model for instruction following. For DPO on math reasoning, we sample 16 responses for each prompt with the model being trained and use the one with the highest reward as the preferred option and the one with the lowest reward as the dispreferred option.

For SFT for math reasoning we train on the Asap7772/cog_behav_all_strategies dataset [8]. For SFT for instruction following, we train on the Smoltalk dataset [12]. For DPO and RLOO for math reasoning we train on the Countdown dataset [1]. For DPO for instruction following we train on the Ultrafeedback dataset [16].

A.2 Comparisons of baseline models

Model	Base models	Number of agent groups	Extractor	Score	Score - at least one correct generation	% at least one correct generation
SFT	Qwen2.5 0.5B Base	N/A	N/A	0.316	N/A	N/A
DPO	SFT	N/A	N/A	0.748	N/A	N/A
RLOO	SFT	N/A	N/A	0.73	N/A	N/A
Untrained agent system	SFT, SFT	1	Agent	0.458	0.674	63.5%
Untrained agent system	SFT, SFT	3	Agent	0.542	0.663	79.0%
Untrained agent system	SFT, SFT	1	Automatic	0.639	0.950	63.5%
Untrained agent system	SFT, SFT	3	Automatic	0.744	0.915	79.0%
Untrained agent system	DPO, SFT	1	Agent	0.779	0.949	80.0%
Untrained agent system	DPO, SFT	3	Agent	0.781	0.943	81.0%
Untrained agent system	DPO, SFT	1	Automatic	0.802	0.989	79.0%
Untrained agent system	DPO, SFT	3	Automatic	0.816	0.978	81.5%
Untrained agent system	RLOO, SFT	1	Agent	0.744	0.940	77.0%
Untrained agent system	RLOO, SFT	3	Agent	0.756	0.939	78.5%
Untrained agent system	RLOO, SFT	1	Automatic	0.775	0.977	77.0%
Untrained agent system	RLOO, SFT	3	Automatic	0.793	0.972	79.5%
Untrained agent system	SFT, DPO	1	Agent	0.536	0.798	63.5%
Untrained agent system	SFT, DPO	3	Agent	0.611	0.750	79.0%
Untrained agent system	SFT, DPO	1	Automatic	0.635	0.943	63.5%
Untrained agent system	SFT, DPO	3	Automatic	0.753	0.926	79.0%

Table 2: We compare a few configurations of our system with the baselines. Agent extraction refers to using the Extractor agent with the calculator, while automatic replaces that with code that extracts and computes the value. For our system, we show the score on all prompts as well as the score on the prompts where the Generator agent successfully outputs at least one correct response. The base models indicate which models were used for the math reasoning and instruction following components of the system.

We compare how well our system does when we vary which baseline models we use as base models for math reasoning and instruction following. Since our agent system performs better without training, we did not fine-tune the system with other base models. For math reasoning, we try our SFT, DPO, and RLOO models as the baseline, and our system shows an improvement relative to those baselines for all of them with 1 agent group and a larger improvement with 3 agent groups.

We also run an experiment to compare our SFT instruction following model with our DPO instruction following model and see a substantial improvement in the version with our Extractor agent. With automatic extraction, the results are approximately the same compared to our SFT instruction following model.

B Implementation Details

B.1 Prompts

B.1.1 Generator agent

```
A conversation between User and Assistant. The user asks a question, and
the Assistant solves it. The assistant first thinks about the reasoning
process in the mind and then provides the user with the answer. User:
Using the numbers %NUMS%, create an equation that equals %TARGET%. You
can use basic arithmetic operations (+, -, *, /) and each number can only
be used once. Show your work in <think> </think> tags. And return the
final answer in <answer> </answer> tags, for example <answer> (1 + 2) / 3
</answer>. Assistant: Let me solve this step by step.
```

B.1.2 Extractor agent

```
The students in the class have a homework problem and wrote out their steps
and final answer. Your job is to extract the final answer. The final
answer starts with <answer> and ends with </answer>. When you extract the
final answer, add " = " after the </answer> so it can be evaluated on a
calculator.

Examples:

User: First, let's try to get close to 86: 17 + 13 = 30 (close to 86)
30 + 56 = 86 (target!)

Let's verify: (17 + 13) + 56 = 86 This works! Let's write it as a formal
expression. <answer> (17 + 13) + 56 </answer> </think> <answer> (17 + 13)
+ 56 </answer>

Extract everything in the last <answer> </answer> (including the <answer>
and </answer> tags too) and remember to add " = " at the end.

Assistant: <answer> (17 + 13) + 56 </answer> =



User: First, let's try some initial attempts: 85 + 34 = 119 (too large)
85 - 34 = 51 (closer to 53) 85 + 2 = 87 (even further away from 53)

Let's try a different approach: 85 - 2 = 83 83 - 34 = 49 49 + 30 = 79
(getting closer but still not there)

Let's try one more time: 34 + 2 = 36 85 + 36 = 121 (too large)

Ah! Let's try: 85 - 30 = 55 55 + 2 = 57 57 - 20 = 37 (getting closer)
</think> <answer> (85 - 30 + 2) - 20 </answer>

Extract everything in the last <answer> </answer> (including the <answer>
and </answer> tags too) and remember to add " = " at the end.

Assistant: <answer> (85 - 30 + 2) - 20 </answer> =

User: %RESPONSE%

Extract everything in the last <answer> </answer> (including the <answer>
and </answer> tags too) and remember to add " = " at the end.
```

```
Assistant:
```

B.1.3 Critic agent

```
The students in the class have a homework problem to use the numbers %NUMS%
exactly once each to create an equation that equals %TARGET%.  Your job is
to evaluate their attempts for correctness.  The 2 criteria are:  1.  The
Numbers must each be used exactly once.  2.  The expression must evaluate
to Target.

If both criteria are satisfied, output "Correct".  Otherwise, output
"Incorrect".

Examples:

User:  Numbers:  [41, 32, 2] Target:  18 Attempt:  <answer>(41 - 32) *
2</answer> = 18 Assistant:  Correct

User:  Numbers:  [46, 71, 12] Target:  37 Attempt:  <answer>71 - 46 -
12</answer> = 13 Assistant:  Incorrect

If each number in %NUMS% is used exactly once and the expression evaluates
to %TARGET%, output "Correct".  Otherwise, output "Incorrect".

User:  Numbers:  %NUMS% Target:  %TARGET% Attempt:  %ATTEMPT% Assistant:
```

B.1.4 Selector agent

```
Your job is to select the first Correct answer from a list of some Correct
and some Incorrect answers.  If none of the answers are Correct, please
select "None of the above".  Each answer will be on a separate line.
Please select the entire answer as demonstrated in the examples.

Examples:

User:
Correct:  <answer>(41 - 32) * 2</answer> = 18
Incorrect:  <answer>41 - 32 - 2</answer> = 7
Incorrect:  <answer>32 * 2 - 41 - 7</answer> = 16
Correct:  <answer>2 * (41 - 32)</answer> = 18

Assistant:  <answer>(41 - 32) * 2</answer> = 18



User:
Incorrect:  <answer>16 / 4 + 9</answer> = 14
Incorrect:  <answer>9 * 4 - 16</answer> = 20

Assistant:  None of the above



User:
Incorrect:  <answer>71 - 46 - 12</answer> = 13
Correct:  <answer>71 - 46 + 12</answer> = 37
Incorrect:  <answer>(41 - 32) * 2</answer> = 18
```

```
Assistant:  <answer>71 - 46 + 12</answer> = 37


Please select the entire first Correct answer.  If none of them are correct,
select "None of the above".

User:  %ANSWERS_WITH_CORRECTNESS%

Assistant:
```

B.2 Loss functions

For the Generator agent, the loss function uses DPO to improve. First we compute the ground truth rewards for each generated response by the agent within the agent group. Then we add a 50% boost to the one that was selected by the agent group (enough to select it if it's the best scoring one, but not enough for an incorrect response to overtake a correct response). Next we use the one with the highest reward as the preferred response and the one with the lowest reward as the dispreferred response for DPO. Like our baseline model training, we use $\beta = 0.1$.

For the Extractor agent, we compute the loss only on responses for which there are valid <answer></answer> tags with an equation (the equation can be correct or incorrect). We use Negative Log Likelihood loss for each response individually, using the automatic extraction answer as the label.

For the Critic agent, we compute the hinge loss for the margin between the logits of the tokens for "Correct" and "Incorrect". We use the ground truth score for the extracted answers as the target labels. We use a margin of 4.

For the Selector agent, we use the first "Correct" answer as the target (if there are no "Correct" answers, we use "None of the above"). We then use Negative Log Likelihood loss comparing the selected answer with the target answer.

B.3 Memory reduction techniques

As described above, we use LoRA to reduce memory during training. We also only keep the one model that is currently being used on the GPU and move the other models to the CPU. For example, during generation, we keep the math reasoning LoRA adapter on the GPU and move the instruction following model to the CPU. Additionally, during training we send each loss backward on the CPU as soon as we compute it (including for each response for the Generator and Extractor agents) instead of aggregating them on the GPU. We also do this during RLOO. After accumulating enough minibatches, we do the optimizer gradient step on the CPU. Instead of computing the logits as we sample, we first sample and then compute the logits by running the sampled response through the model again.

B.4 Evaluation

For evaluation, we first use vLLM to generate up to 12 responses for each prompt (4 per agent group and up to 3 agent groups) using the math reasoning model. For the trained models, we compute the responses separately for each agent group by using the trained LoRA adapters. Next we run the prompts through our system in Generate mode to skip loss function computation and use the vLLM responses instead of the Generator agent and then compute the score of the selected response. As described above, we use up to 1024 tokens during evaluation for generation.