CIQA : a Coding Inspired Question Answering model

Anonymous ACL submission

Abstract

Generating code to solve question-answering (QA) problems can help scale up analyses or workflows that may be too time-consuming or complex to do manually. This can be especially useful in scientific applications, where large datasets and complex analyses are common. Natural language approaches converting texts describing processes to code showed initial success in reasoning over such textual problems. The limitations of existing text-to-code models are evident when attempting to solve QA problems that require knowledge beyond what is presented in the input text. We propose a novel domain-agnostic model to address the problem by leveraging domain-specific and open-source code libraries. Our model learns to inject this knowledge into the code generation process given a textual QA problem. Our study demonstrates that our proposed method is a competitive alternative to current state-of-the-art models, with the added capability of solving problems beyond the scope of their capacity with high accuracy. We also present a new QA dataset, focusing on scientific problems in the domains of chemistry, astronomy, and biology, for the benefit of the scientific community.

1 Introduction

The challenge of question answering has been extensively explored, with efforts made to improve traditional sequence-to-sequence(Seq2Seq) models for generating accurate responses to complex questions (Anil et al., 2023).

In scientific domains, answering even seemingly simple questions often requires substantial data and inference capabilities. Texts in these fields often detail intricate processes with multiple variants and question outcomes under different conditions, and we cannot assume the model has encountered all entities or question types (Peretz et al., 2023). Addressing such questions necessitates understanding the interactions among the entities involved and simulating their state transitions during the process execution under different conditions, which requires external world-knowledge.

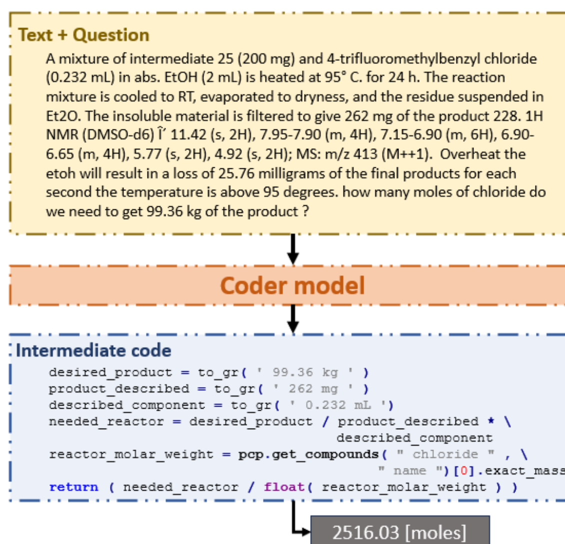

Figure 1: Question example that illustrates the use of the `pubchempy` Python library

Recently, it has been shown that natural language to some logical form has achieved superior results for inference and reasoning over seemingly simple text datasets (Berant et al., 2014; Kiddon et al., 2016; Tandon et al., 2019; Peretz et al., 2023). Although impressive results have been demonstrated on a scientific QA dataset(Liu et al., 2023), all code generation models for question-answering (Peretz et al., 2023; Chen et al., 2021; Soliman et al., 2022; Chen et al., 2022, 2020; Zhang and Moshfeghi, 2022) and even foundation models (Brown et al., 2020; OpenAI, 2023), struggle with questions requiring domain-specific knowledge and external data not present explicitly in the question itself or previously seen explicitly.

Consider the following example:

Text: "We have **3 liters** of **water**, if the temperature of the liquid exceeds 100°C,

1 gram of water will evaporate every second."

Question: "How much time should we wait until we have 20.03 moles of vapor if the temperature is $212.1°F$?"

A straightforward calculation and lookup would reveal that water evaporates at a temperature of $100°C$. Additionally, recognizing that $100°C$ is equivalent to $212.1°F$, and understanding the chemical composition of water, allows us to conclude that the molar mass of water is $18.01528 g/mol$. These operations and knowledge, though seemingly simple, are essential for simulating the process and crafting the corresponding process code to answer the question.

In this work, we introduce a method that allows the injection of external world knowledge and querying numerous databases in various fields by training the model to utilize external scientific code libraries during code generation (APIs) as tools using the library documentation. We propose a model designed to effectively incorporate library calls into the generated code by using the library documentation and open-source code thereby allowing the model to incorporate APIs in the generated code and granting access to data that is not explicitly stated within the question's context. We aim to refine semantic parsing by employing open-source code repositories to produce general-purpose code. The unique incorporation of "external data" into intermediate code via APIs and code libraries enriches the semantic parsing process, increasing the capacity to generate more intricate and accurate code. This enhancement not only advances semantic parsing but also aims to mitigate the reliance on extensive NL-code data. To that end, we designed a first-of-its-kind API-attention mechanism that can be applied in many applications as a tool attention mechanism. We employ reinforcement learning to boost the accuracy of final answers, opening a new frontier in the field of question-answering.

We evaluate our algorithm compared to state-of-the-art (SOTA) approaches for QA and foundation models, and present superior results on a scientific dataset.

The contributions of this work are threefold:

1. We present a novel framework for incorporating external knowledge by leveraging API calls during code generation to simulate textual processes for QA. And introduce a first-of-its-kind external attention mechanism to aid the model with APIs integration.
2. We present a dataset for simulation questions of scientific processes in scientific fields: astrophysics, chemistry, and biology. We contribute it and our code to the community for further research.
3. We perform an extensive evaluation of our algorithm compared to SOTA and demonstrate superior results achieving a 59% exact answer accuracy compared to 34% reached by fine-tuned foundation models and 23% with text-to-code generation models for QA.

The subsequent sections of this paper are organized as follows: Section 2 will review existing research in the field, positioning our paper's contribution within this context. In Section 3, we define the problem we aim to address, outlining our research objectives. Section 4 will provide a comprehensive discussion of our model's architecture and training methodology. Furthermore, in Section 5, we will discuss baseline models, the process of dataset construction, and the methodology employed for evaluation. Lastly, in Section 6, we will present an in-depth analysis of CIQA's performance relative to other state-of-the-art models, including an ablation study and qualitative evaluation.

2 Related work

Question-answering is a prominent area of research that has been tackled with diverse methodologies. These approaches can be broadly divided into two categories when no external database of answers exists: large language models (LLMs) trained on question-answer aligned datasets and semantic parsing approaches.

Recently, there has been significant progress in the field of semantic parsing-based methods (Zhang and Moshfeghi, 2022; Chiang and Chen, 2019). These techniques aim to convert text into an intermediate logical form in order to effectively answer questions. As a result, they have achieved state-of-the-art results. Notably (Peretz et al., 2023) have significantly contributed to this area. In their work, they have chosen to represent the posed question using a domain-specific code. This approach has led to impressive outcomes, particularly in the context of chemical texts that describe processes.

Our research falls within this realm of semantic parsing-based methods. However, our contribution

lies in the incorporation of external data through the utilization of APIs. We achieve this by leveraging open-source code samples and publicly available documentation. This approach allows us to generate a general-purpose code that is capable of addressing questions in complex scientific domains such as chemistry, biology, and astronomy (see Fig. 1), with less training data.

A parallel line of research has focused on generating code from given texts. Described as "co-pilots" these works primarily assist engineers in coding tasks. These models focus on texts describing the code to be generated. Our work diverges from this trend by focusing on interpreting and processing text that details complex processes and poses a question regarding this text.

Recent advancements have witnessed a shift towards the integration of external knowledge into LLMs for textual reasoning (Zhang et al., 2023; Geva et al., 2020). Many of these models access human-like reasoning tools, such as calendars and calculators (Schick et al., 2023), and heavily depend on domain-specific data and tools. We introduce what we believe to be the first domain-independent model for QA focusing on the generation of code using external APIs.

3 Problem Definition

We introduce the following notations: the input text prompt is represented as $\sigma \in \Sigma$, the expected final answer as $y \in \mathcal{Y}$, the intermediate code as $c \in \mathcal{C}$, and a similarity metric as d. We define two categories of generation models: (1) generative Seq2Seq models denoted as $\pi_a \in \Pi_a$, which aim to generate answers to the prompt and optimize the objective:

$$max_{\pi_a \in \Pi_a} \Big(d\big(\pi_a(\sigma), y\big) \Big) \quad (1)$$

(2) Code generation models that produce code as an intermediate step before generating an answer, denoted as $\pi_c \in \Pi_c$, where the objective is:

$$max_{\pi_c \in \Pi_c} \Big(d\big(\texttt{exec}(\pi_a(\sigma)), y\big) \Big) \quad (2)$$

This study focuses on problem instances that can be solved through an auxiliary task, with the objective of maximizing Equation 2.

4 CIQA

In the pursuit of advancing the efficiency and intuitiveness of question-answering systems, the research community has explored a diverse range of methodological approaches. Motivated by the foundations of programming, we present a novel architecture termed the Coder Inspired Question Answering, abbreviated as CIQA (pronounced as "seeker"). By combining the principles of coding with the nuances of natural language processing, CIQA endeavors to reconcile structured programming logic with the intricacies of human language. Capitalizing on tools and methodologies familiar to the programming domain, CIQA demonstrates state-of-the-art performance, underscoring the potential of integrating coding methodologies to augment the precision of question-answering frameworks.

Diverging from conventional methods, CIQA obviates the need for retraining on paired data of texts and codes to accommodate new APIs. Instead, it draws inspiration from the approach taken by software engineers who, when encountering a challenge, often resort to studying open-source code, and the documentation to learn the utilization of APIs from accessible libraries. Similarly, CIQA acquires insights into API usage from open-source code, allowing for smooth integration into its assigned task, without the need for large amounts of aligned training data.

The CIQA architecture, shown in Fig 2, uses an encoder-decoder framework. Details of the encoder and decoder can be seen in Fig 3. This architecture termed the "coder" model, takes in text and a question to produce executable code that, when run, answers the question. Additionally, CIQA features an API-attention component, essentially an NL encoder. This component processes API documentation to determine its relevance to the question. The result from the API attention then influences the next token prediction in the "coder" model.

4.1 Model Training

The training process for our model is divided into two primary stages: the code generation stage, and the final answer generation stage.

The integration of the two stages fosters the development of a robust and flexible code generation engine within CIQA. The result model demonstrates an exceptional capability to produce high-quality code solutions, tightly aligned with the question posed to the textual input.

4.1.1 Generating Code

A dataset composed of paired prompts and corresponding Python code is used for training. This

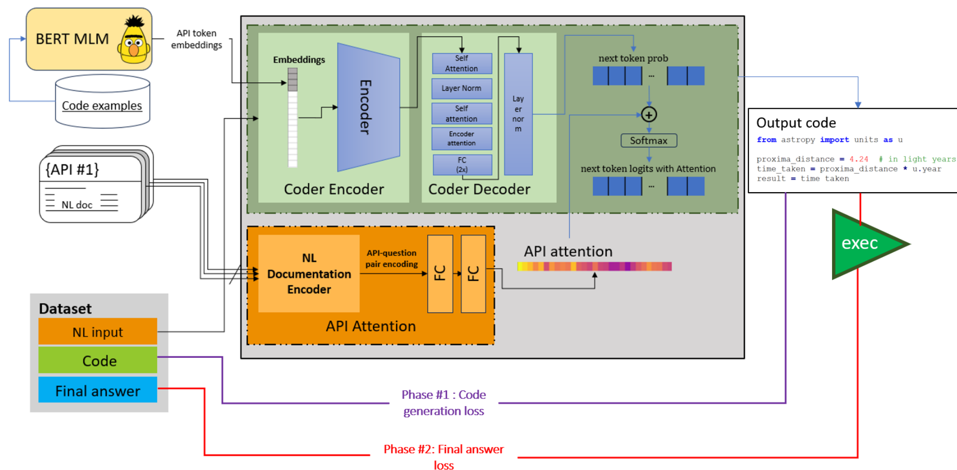

Figure 2: CIQA Architecture: the green component, denoted the *coder* model is a model responsible for generating the code, such that combined with the function of the API attention component, produce relevant code to the domain of the question as seen in Eq 3. The API attention component, the orange component, takes as input an NL input that consists of the the context-question duo, and the API documentation see more in subsection 4.1.4.

code solves the designated problem, leading to the generation of the target answer when executed. In this phase, the objective is to optimize the synthetic similarity between the reference code and the code generated by the model. Specifically, we focus on enhancing the BLEU score as can be seen in Alg 2, a widely recognized metric for evaluating the coherence and alignment between the two text sets, that we applied to both the generated code and the gold standard code.

4.1.2 Leveraging code repositories

To incorporate a specific library into our model, we first need to obtain relevant embeddings for its function calls. We do this by searching code repositories for instances where the library is used, giving us insights into its applications and potential uses.

Our next steps involve pre-processing: We remove non-essential text such as documentation and exclude functions not using the target library. We also standardize all examples in the dataset, ensuring that all code samples import the APIs under the same name. We then extend the model's tokenizer by adding a token per API.

To integrate API embeddings into our model, we train the embedding layer of the *coder* model using BERT for masked language modeling task, where some input tokens are replaced with a `[MASK]` token, and the model predicts these masked tokens using the surrounding context. While we used BERT for its straightforwardness, other Transformer models can be used.

Finally, the updated embedding layer is merged back into our original model, equipping it with the necessary API embeddings.

4.1.3 Generating Final Answer

In the prior phase, our objective was to optimize the generation of intermediate code. In this phase, our focus shifts to ensuring the code produces the correct final answer.

To ensure the code's output closely matches the desired answer, we adopt a method inspired by (Keneshloo et al., 2020; Wu et al., 2017), which has shown top-tier results in similar tasks. We employ a Reinforcement Learning (RL) approach to optimize the model. It's worth noting that we don't have a reference intermediate code; we only have the final answer. Our reward function, detailed in Alg 1, emphasizes both the successful compilation of the code and its ability to produce the correct final answer.

This dual focus ensures the generated code is both syntactically correct and functionally effective, enhancing the model's ability to produce code that meets both synthetic and semantic requirements.

The use of RL is expected to allow the model to find varied solutions to problems, even without relying on reference code, thus improving its adaptability in using APIs in diverse situations.

In the reward function, we utilized BLEU score, which serves several key purposes. It is employed to measure similarities between various types of outputs, including numerical, logical answers (true and false), and multiple unordered outputs, thus

evaluating the model's capability to generate code that provides accurate responses across different scenarios. Furthermore, the BLEU score captures similarities in answers on different orders of magnitude, an essential aspect when handling values that differ significantly due to various units in the question. Unlike the L2 norm, which may impose harsh penalties for unit mix-ups, the BLEU score does not heavily penalize such discrepancies, recognizing them as potential pitfalls but not severe errors. This approach allows us to understand and evaluate the model's handling of such cases without excessively penalizing these differences, while still aiming to avoid them if possible.

Algorithm 1 Reward calculation

Input: c_{pred} , c_{ref}
1: **if** c_{pred} does not compile **then**
2: **return** 0
3: **end if**
4: bleu = BLEU$\big($`exec`(c_{ref}), `exec`$(c_{pred})\big)$
5: reward = 1 - 1/bleu
6: **return** reward

4.1.4 API attention

We utilized a natural language encoder transformer model (Dunn et al.) to encode the natural language input. The model we employed takes an API documentation file, as well as a context-question pair as input. The encoder's output is passed through two fully connected layers with sigmoid activation and generates a match score as we can see in Fig 2. This match score indicates the relevance of the API to the given question. The match score is incorporated into the probability calculation for the next token in the coder's decoder by adding the output match score to the probability of the API token in the coder's decoder model.

During training, this component received positive reinforcement whenever the executed code produced the desired outcome as specified in Alg 3. Furthermore, during the code generation phase, the loss was propagated through this component.

Given a decoder's next token prediction vector $\mathbf{p}$ with entries p_i, the attention model's output as $\mathbf{a}$, we define new next token predicted probability vector d:

$$d_i = \begin{cases} p_i + a_i & \text{if } i \text{ is an API token} \\ p_i & \text{otherwise} \end{cases} \quad (3)$$

Algorithm 2 Code generation training

Input: π_c , dataset $\in (\Sigma \times \mathcal{C})$
1: $batch \leftarrow []$ ▷ Initialize an empty list
2: $model \leftarrow \pi_c$ ▷ pre-trained coder model
3: **for** each pair (σ_i, c_i) in dataset **do**
4: $o_i \leftarrow \pi_c(\sigma_i)$
5: $loss \leftarrow \mathcal{L}(o_i, c_i)$
6: $\pi_c.backprop(loss)$
7: **end for**

Algorithm 3 Final answer generation training

Input: π_c, $dataset \in (\Sigma \times \mathcal{Y})$
1: $model \leftarrow \pi_c$ ▷ pre-trained coder model
2: **for** (σ_i, y_i) in dataset **do**
3: $o_i \leftarrow \pi_c(\sigma_i)$
4: **if** y_i does not compiles **then**
5: $reward \leftarrow \mathcal{L}.min_reward$
6: **else**
7: $y_pred \leftarrow$ `exec`(o_i)
8: $reward \leftarrow \mathcal{L}(y_pred, y_i)$
9: **end if**
10: $\pi_c.backprop(loss)$
11: **end for**

4.2 Implementation Details

To implement our base models we trained our model on the CoNala dataset (Yin et al., 2018) until the validation error began to increase. The API embeddings were learned by using a BERT for MLM, such that we took the learned embedding layer from the embedding layer in BERT and inserted it into the coder model. The coder model was then fine-tuned for 12 epochs on the Seq2Seq data. The training process employed an initial learning rate of $lr = 1e - 5$. The Seq2Seq trainer from the Hugging Face library was utilized for this purpose.

5 Empirical Evaluation

In this section, we describe our empirical methodology.

5.1 Datasets

In this study, a new dataset was constructed, "SeKq": "Scientific external knowledge question dataset", which was assembled by experts from the fields of biology, astronomy, and chemistry, in collaboration with computer scientists. These domain experts formulated the questions, while the computer scientists developed the code to address them. The dataset encompasses context, questions, definitive

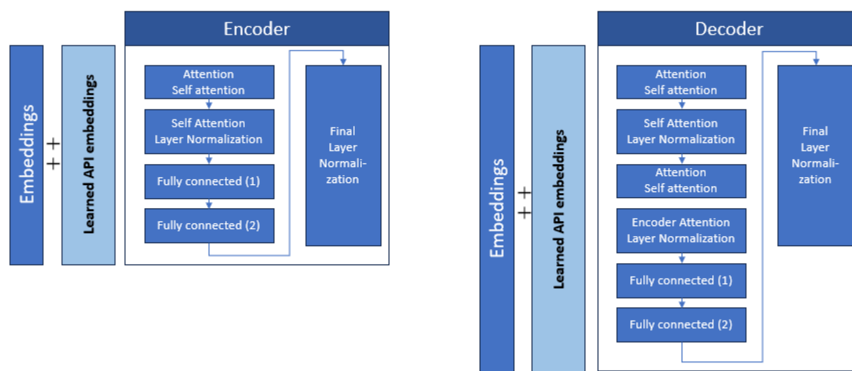

Figure 3: Encoder and decoder internal architecture: the learned embeddings of the APIs are added to the embeddings directory of the encoder and decoder.

Domain	constants	APIs
Biology	0.1	2
Astrophysics	2.02	1.14
Chemistry	0	1.42

Table 1: Analysis of average API and constant usage across different fields

answers, and the intermediary code that yields the desired response.

Library documentation relevant to each domain was gathered. This led to the creation of a dataset that includes more than 30 unique libraries capable of addressing the questions and facilitating data structuring and reasoning. While not all these libraries are utilized in the reference code, their inclusion was important to provide the model with a comprehensive range of potential solutions, which can be further expanded and allow the model to answer an even wider spectrum of questions.

We conducted an analysis to evaluate the use of the required APIs in the gold standard code in each domain as can be seen in table 1.

Each question was created to require the model to retrieve external values or possess prior domain knowledge, which isn't explicitly formatted within the question itself.

5.2 Baselines

In our study, we conducted a comparative analysis between our proposed method and SOTA natural language (NL)-code synthesis models. We compared our model against much larger models that trained on much more data. MarianNMT (Junczys-Dowmunt et al., 2018), which was trained on NL-to-code generation tasks from the CoNaLa dataset (Yin et al., 2018), recently reached SOTA results.

We also compare with Foundation generation models. We select GPT-3.5(Brown et al., 2020) and GPT-4(OpenAI, 2023) models. These models are known for their advanced capabilities in language processing and generation. By contrasting our base model with these larger models, we aimed to demonstrate the effectiveness and competitiveness of our proposed training technique. We also experimented with CoT prompting (Wei et al., 2023) and GPT-4 with code interpreter versions.

Additionally, we compare our model with the SOTA for QA, LLama-QA a LLama 2 7B (Anil et al., 2023) and the SOTA for QA using code generation as an auxiliary task (Soliman et al., 2022).

5.3 Evaluation Metrics

In our study, we employed an exact-match evaluation metric on the code's output to assess the performance of the different models. This metric measures the degree of exact correspondence between the generated code's output and the target answer. By utilizing an exact-match evaluation, we aim to evaluate the precision and accuracy of our model in generating code that precisely produces the expected answer. This evaluation approach provides a stringent criterion for assessing the fidelity of our model's code generation capabilities and enables a precise comparison of results.

6 Empirical Results

In this section, we present our empirical results and ablation tests, we also performed a study on how each component contributed to the success of our model.

Algorithm Type	Model	Accuracy
SOTA code generation	finetuned marianCG	0.23
Foundation Models	GPT3.5 finetuned with final answer	0.34
	GPT4 no finetune	0.27
	GPT-4 with code interpreter	0.30
	GPT-4 with Change of Thought	0.31
SOTA QA models	LLama-QA	0.24
SOTA for QA using code generation	NPS-SQA	0.32
CIQA (Ours)		**0.59**

Table 2: Main result table: comparison between different SOTA algorithms on our dataset.

6.1 Main Result

We can see in Table 2 that our model outperforms all of the other models by a large margin. In our dataset, our model performs better than the second-best model by 26%. We hypothesize that the gap in performance is attributed to the effective use of APIs as can be seen in sec 6.2.1.

We also see a significant improvement versus other code generation models for question answering. We attribute that to our final answer generation phase, where we consider the output of the generated code and tend to maximize the match between the output and the target answer.

In the results presented in Table 2, our approach exhibits superiority over competing models. While existing methods primarily focus on optimizing the generation of analogous final answers, our model facilitates a deeper interpretation of the question, ultimately leading to the determination of the final answer to the posed question.

6.2 Ablation Tests

6.2.1 Impact of fine-tuning training size on performance

To assess the impact of the size of natural language-to-code (NL-code) training data on our model, we formulated and executed an experiment in two distinct configurations:

1. In the initial configuration, we reserved 50% of the dataset entries specifically for natural language-to-code training. The remaining 50% allowed the model to access only the problems expressed in natural language, along with the corresponding expected solutions. In this context, our model demonstrated an accuracy of 0.45, more can be seen in Fig 4.

2. In the subsequent configuration, we deployed the entire dataset, covering 100% of the entries, limiting the model's exposure to just the natural language-to-code data. Under this restriction, the model's accuracy amounted to 0.59.

These findings highlight an essential observation: the conversion of a question into code transcends the simplicity of a Seq2Seq task. It shows that considering the final output of the model could help the model learn to use code to achieve its target.

6.2.2 Impact of embeddings on CIQA performance

We also conducted an experiment to see how our model benefited from the learned API embeddings. We trained our model in the same manner, but this time we did not introduce the new API embeddings from the code repositories, our model performed 0.48 accuracy against the 0.59 that we saw earlier.

6.2.3 Impact of final answer generation on CIQA performance

We trained the base coder model with only the generating code phase, without the RL phase we arrived at an accuracy of 0.46. This emphasizes the role the final answer generation phase has in generating a higher-quality code. We noticed that without the final answer generation phase the model could write good code, as it failed in assigning the correct parameters to the API calls.

6.2.4 Impact of the API attention on CIQA performance

To see the effect the API attention on the performance of the model we trained the model in the same manner as we did before, but his time without the API attention mechanism. We noticed that when trained on small datasets as can be seen in Fig As expected we noted that the model did not always use the correct API in the desired place which led to worse results.

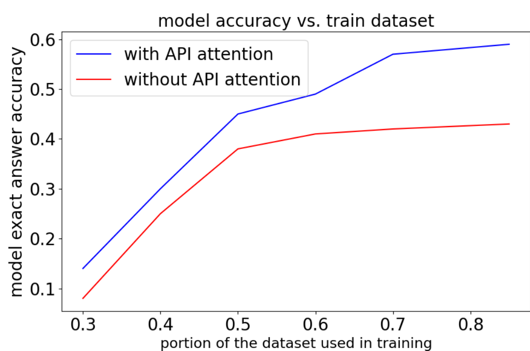

Figure 4: Exact match accuracy with and without the API-attention mechnanism

6.3 Qualitative Evaluation

In this section, we provide qualitative examples of the performance of our algorithm.

6.3.1 Strengths of CIQA

Through a comprehensive analysis, we identified specific questions where our model demonstrated significant strengths. Particularly with respect to boolean questions. In our dataset, boolean questions necessitated fewer lines of code and reduced utilization of APIs. Consequently, our model encountered fewer pitfalls. Through our analysis, it became evident that our model outperformed competing models on these simple questions within the framework, achieving an accuracy of 0.59. In contrast, the second-best model, the fine-tuned GPT3.5, demonstrated comparable performance across both categories of questions, yielding an accuracy score of 0.34.

6.3.2 Limitations of CIQA

CIQA has demonstrated substantial capabilities in answering questions through code generation with API-enabled external data access. However, we recognize a weakness when handling complex compounds: The CIQA's success depends on the accurate identification of compound names within the API's database. For example, "10% palladium" consists of 10% of by weight of palladium and the rest is carbon, ideally, the model should have decomposed the alloy into the consisting compounds, and used the API to fetch the needed information about them separately.

7 Conclusions

Our research delved into the realm of scientific question-answering, focusing on Chemistry, Astronomy, and Biology. We noticed that conventional models often failed when faced with complex questions that required data outside of the span of the context. In response, we crafted a unique method that generated code to simulate the processes to generate answers. A common obstacle with many models is their struggle to incorporate specialized knowledge during these simulations. To address this, we pioneered a method that seamlessly merges external APIs into the code creation process. This lets the model tap into crucial information beyond the immediate context of the question, even with just a modest amount of relevant training data. By employing an encoder-decoder structure, our method surpasses the performance of current leading techniques, even when data is scarce. We're proud to present a novel framework to seamlessly embed external knowledge into the answering process, a carefully curated dataset by experts, and evidence of our model's enhanced accuracy. Looking ahead, we believe there's potential to broaden our method's application beyond just the sciences. It would be worthwhile to explore the API integration as a general tool and train the model to better utilize these tools in answering questions.

References

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. arXiv preprint arXiv:2305.10403.

Jonathan Berant, Vivek Srikumar, Pei-Chun Chen, Abby Vander Linden, Brittany Harding, Brad Huang, Peter Clark, and Christopher D. Manning. 2014. Modeling biological processes for reading comprehension. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1499–1510, Doha, Qatar. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In Advances in Neural Information Processing Systems, volume 33, pages 1877–1901. Curran Associates, Inc.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan,

Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V. Le. 2020. Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension. In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net.

Ting-Rui Chiang and Yun-Nung Chen. 2019. Semantically-aligned equation generation for solving and reasoning math word problems. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 2656–2668, Minneapolis, Minnesota. Association for Computational Linguistics.

Matthew Dunn, Levent Sagun, Mike Higgins, V. Ugur Guney, Volkan Cirik, and Kyunghyun Cho. Searchqa: A new q&a dataset augmented with context from a search engine.

Mor Geva, Ankit Gupta, and Jonathan Berant. 2020. Injecting numerical reasoning skills into language models. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 946–958, Online. Association for Computational Linguistics.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. In Proceedings of ACL 2018, System Demonstrations, pages 116–121, Melbourne, Australia. Association for Computational Linguistics.

Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K. Reddy. 2020. Deep reinforcement learning for sequence-to-sequence models. IEEE Transactions on Neural Networks and Learning Systems, 31(7):2469–2489.

Chloé Kiddon, Luke Zettlemoyer, and Yejin Choi. 2016. Globally coherent text generation with neural checklist models. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 329–339, Austin, Texas. Association for Computational Linguistics.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning.

OpenAI. 2023. Gpt-4 technical report.

Gal Peretz, Mousa Arraf, and Kira Radinsky. 2023. What if: Generating code to answer simulation questions in chemistry texts. In Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '23, page 1335–1344, New York, NY, USA. Association for Computing Machinery.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

Ahmed S Soliman, Mayada M Hadhoud, and Samir I Shaheen. 2022. Mariancg: a code generation transformer model inspired by machine translation. Journal of Engineering and Applied Science, 69(1):1–23.

Niket Tandon, Bhavana Dalvi, Keisuke Sakaguchi, Peter Clark, and Antoine Bosselut. 2019. WIQA: A dataset for "what if..." reasoning over procedural text. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 6076–6085, Hong Kong, China. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Lijun Wu, Li Zhao, Tao Qin, Jianhuang Lai, and Tie-Yan Liu. 2017. Sequence prediction with unlabeled data by reward function learning. In International Joint Conference on Artificial Intelligence.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In International Conference on Mining Software Repositories.

Jiaxin Zhang and Yashar Moshfeghi. 2022. ELASTIC: Numerical reasoning with adaptive symbolic compiler. In Advances in Neural Information Processing Systems.

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. Toolcoder: Teach code generation models to use api search tools.