# **Controlled Experiments as a Comparison Between Base and Derived Classes**

Shiro Takagi<sup>1</sup>

<sup>1</sup>Unktok shiro.takagi@unktok.com

## Abstract

In the journey toward automated machine learning research, experimental code templates play a crucial role. Template design involves a fundamental trade-off: specific templates aid complex experiment generation but limit flexibility, while abstract templates offer flexibility but provide less support for complexity. Thus, designing templates at an appropriate abstraction level is essential. This short conceptual paper presents an idea of an experiment code template design in machine learning research automation. We propose representing controlled experiments as comparisons between base and derived classes, where the control group is implemented as a base class representing typical machine learning workflows, and experimental interventions are implemented through inherited classes with targeted method overrides. While our current framework has several limitations, our demonstration examples show its potential to provide a flexible and abstract framework for ML research automation while preserving the structure needed for complex experiment automation. Code is available here: https://github.com/t46/ml-control-experiment Note: This report is a conceptual paper, presenting ideas along with a few preliminary examples rather than a technical paper with solid experimental supports.

## Introduction

Machine learning research automation has made significant strides in recent years, showing a potential for a future pivotal shift in how we approach scientific discovery in this field (Lu et al. 2024; Ifargan et al. 2024; Li et al. 2024b; Weng et al. 2024; Huang et al. 2024). As we conceptualize research as the process of proposing and validating new ideas and discoveries (Schickore 2022), the automation of machine learning research necessitates the automation of not only discovery but also verification; especially both automated planning and execution of experiments.

Previous work in this domain has primarily focused on representing experiments as Python scripts, with automation achieved through the generation, modification, and execution of these scripts (Lu et al. 2024; Li et al. 2024b; Takagi, Yamauchi, and Kumagai 2023). While the long-term vision will be to generate human-level research code completely from scratch, current limitations in experiment code generation capability of AI require carefully crafted experimental template codes to produce high-quality experimental implementations (Lu et al. 2024). For instance, you may need ex-



Figure 1: Trade-off in experiment code template design: specific code templates enable complex experiment implementations but limit flexibility, while generic templates offer broader applicability at the cost of functionality. The challenge lies in finding the right balance between specificity and adaptability in template design.

plicit specification of datasets, metrics, and model architectures within their templates.

As an intermediate step in transitioning from fully specified templates to a state with minimal or no predefined structure, it is crucial to increase the abstraction level of the information contained within these templates. While specificity in experimental code templates helps AI generate complex experiment code, too much detail reduces flexibility, making it harder to adapt the templates to diverse research ideas and research topics. For example, an experimental code template designed to validate the effectiveness of a newly proposed neural architecture for image classification would not only be unhelpful for an experiment aimed at validating the effectiveness of a new prompt design for text generation, but it might even introduce unintended biases when the AI generates code. Thus, designing templates at an appropriate abstraction level is essential for enabling AI to flexibly generate experimental code that can adapt to different research ideas and contexts.

This paper proposes a conceptual framework for designing machine learning experiment code templates based on three interconnected principles. First, we adopt the view of representing machine learning processes as abstract workflows, as is done in fields such as MLOps. We consider a machine learning process as a sequence that encompasses model and data initialization, model training, and prediction generation using the updated model.

Second, we frame controlled experiments in machine learning research as a process involving two workflows and a single evaluation criterion. One workflow represents the control group, while the other represents the experimental group with an added intervention—new proposal in ML research. The evaluation criterion then serves to assess the effectiveness of this intervention, measuring the impact of the proposed approach against the control baseline.

Third, we propose expressing experimental group workflow through inheritance from a control group workflow. By using object-oriented principles, experimental variations are implemented as inherited classes that modify or extend a base workflow representing the control condition. This approach is expected to provide a more flexible and abstract framework for representing experimental designs while maintaining the structure necessary for automated execution.

In this short report, we first detail this idea of experiment code template design. Then, we explain a potential of this idea as flexible but moderately specific code template with subjective examination against few examples generated by a prototype system following the proposed idea. The point of this conceptual paper are summarized as follows:

- We discuss the potential role of template code in the automation of machine learning research and emphasize the importance of designing template code with an appropriate level of abstraction.
- We abstractly describe the machine learning workflow as a process of modifying a learner using data and then applying the modified learner to the data.
- We propose an idea of flexible template code design for machine learning experiments by abstractly describing a base class that represents a control group in a machine learning workflow. Experimental groups are created through subclassing and method overrides, facilitating comparative evaluation as part of a controlled experimental process in machine learning.
- Using a few generated examples, we demonstrate the feasibility of conducting controlled experiments with proposed approach. We show that the proposed template design has a potential to make complex experiment implementations easier compared to having no templates, while maintaining greater flexibility than domain-specific templates.

# **Proposed Idea**

To effectively automate machine learning experiments, you must carefully abstract and organize the process of both "experimentation" and "machine learning." In this section, we present an approach to characterizing these elements.

# **Machine Learning Workflow**

We recognize the challenge of creating a universally comprehensive characterization of all machine learning approaches and acknowledge that more refined definitions exist in fields such as MLOps and theoretical machine learning research. Nonetheless, we propose a working characterization to serve as a foundation for future refinement.

In one view, machine learning can be conceptualized as a process involving a learner and data, where the learner is trained on a subset of the data and subsequently generates outputs for previously unseen data. In other words, we represent the machine learning process here as a workflow consisting of two sub-processes: model training and model inference, each taking the model and data as inputs. For simplicity, we refer to this workflow as the *MLWorkflow* in this paper. A high-level illustration of MLWorkflow is shown in Fig. 2.

This characterization is intentionally abstract, providing a flexible template that can encompass a broad range of machine learning research. The process introduced here seems to be widely applicable across different machine learning research domains. For example, while image generation and text generation research may differ in the content of their models and data, they both share the common requirement of needing a model and data. At the same time, in both research areas, it seems common to require either training or running model as part of validating the proposal.

On the other hand, since the content of the data and model naturally differs, a template code specialized for image generation would likely not be useful for text generation. Moreover, even if we restrict the focus to text data and LLMs, a template designed for studies on LLM mechanistic interpretability may not suit research aimed at proposing new language model architectures. This is why a template code that describes the machine learning process in an abstract way is considered important.

# **Characterizing Control Experiment**

Before presenting an abstract description of machine learning experiments, we will first briefly review general concepts related to experimentation. Although experiments can take various forms, our focus here is on controlled experiments, a type commonly adopted in machine learning research.

At its core, an experiment is an empirical effort to test hypotheses or ideas, carrying inherent uncertainty since the validity of these ideas is unproven. In a controlled experiment, this testing involves comparing two groups: an experimental group that receives an intervention and a control group that does not. In such comparisons, it is essential to carefully define what is being compared and how. For example, if an intervention is expected to improve the reliability of model outputs, reliability should be measured with appropriate metrics, and these scores should be compared between the two groups.

Therefore, controlled experiments necessarily comprise three essential components: (1) a control group, (2) an experimental group (the control group plus an intervention), and (3) an evaluation method that defines what to evaluate and how. The validity of the intervention is verified by comparing components (1) and (2) in terms of component (3).

# **Representation of ML Experiments**

Building on our previous abstractions of machine learning and experimentation, we can understand controlled experi-



Figure 2: MLWorkflow: Abstract typical workflow of machine learning. In much machine learning research, both data and a learning model are required. Part of the data is used as training data to train the model, and the trained model is then applied to data not used in training. In actual machine learning research, both data preparation and model preparation are involved, so while these are essential, treating them merely as "inputs" may not be appropriate. Please note that what is presented here is just a simplified representation of machine learning process and further refinement is necessary.

ments in machine learning research as processes that compare two versions of MLWorkflow - one with an intervention and one without - using specific evaluation criteria to validate the intervention's effectiveness. That is, in this framework, (1) the control group is represented by the basic machine learning workflow, (2) the experimental group is the workflow with modifications, and (3) the evaluation method determines how to assess the validity of these modifications.

Here, note again that "intervention" corresponds to a new proposal in the research and the evaluation method assesses the proposal's validity, not just the model's performance. For example, when proposing a new neural architecture, the intervention would involve replacing the baseline model in the existing workflow with the proposed architecture, with classification accuracy serving as the evaluation metric. Similarly, when proposing a faster-converging optimizer, the intervention would involve replacing the existing optimizer, and the evaluation might measure the number of steps required to reach a specific loss threshold.

**Code Representation of Controlled Experiments in ML Research** We now propose a specific approach to expressing these elements in code. The key point is that controlled experiments require maintaining identical conditions between groups except for the intervention. We suggest that this requirement aligns naturally with the concepts of class inheritance and method overriding in object-oriented programming.

Class inheritance allows a derived class (child class) to inherit functionality and properties from an existing class (parent or base class). Method overriding enables the derived class to replace specific methods from the base class with new implementations while maintaining the same method name. Therefore, we can represent the control group with a base class and create the experimental group by inheriting from this base class and overriding specific methods to implement the intervention. Specifically, we define an MLWo rkFlow class to represent the machine learning workflow and create a NewMLWorkFlow class that inherits from it, implementing research proposals through method overrides.

The use of inheritance offers particular advantages in the context of contemporary code generation and modification using Large Language Models (LLMs) since current LLMs struggle to follow strict rules. When modifying code for controlled experiments, we need LLMs to create experimental groups while preserving the control group code. By managing base and derived classes in separate files, we can provide the base class as context while limiting LLM modifications to the derived class. This approach allows us to create experimental groups that inherit functionality from the control group without risking modifications to the control implementation.

Having established our representation of control and experimental groups, we turn to the expression of evaluation methods. We can conceptualize evaluation methods as functions that take "outputs" from both groups as input and produce validation results. Importantly, these "outputs" are not necessarily limited to model predictions. For instance, in the optimizer comparison example introduced above, we need both "loss values" and "step counts" relative to specific thresholds in order to measure the number of steps required to reach a loss value. In other words, what serves as the input to the evaluation function in the MLWorkflow can vary depending on what aspect is intended to be evaluated. Therefore, the evaluation method's input is more accurately described as "logs of intermediate artifacts generated during workflow execution that are necessary for validation." We should therefore design evaluation functions to receive intermediate artifacts from two machine learning workflows as input and produce comparison results as output.

In summary, we propose representing controlled experiments in machine learning research through an objectoriented paradigm: the control group is defined as a base class, the experimental group inherits from this base class with specific interventions, and the comparison between these classes is evaluated using targeted performance metrics receiving the logs of intermediate artifacts from the both workflows. For the convenience, we call this framework *Controlled Experiments as Represented by Inheritance* (*CERI*) framework in this paper.

# Implementation

Having explained the high-level concepts of our proposal, we will now describe the implementation of this framework for the current demonstration.

# Overview

We prepared a main script called run\_controlled\_expe riment.py (provided in the GitHub repository) to execute controlled experiments, which is represented as the combination of experiment.py and mlworkflow.py.

First, the main script generates the research idea, which comprises two components: the research context and the proposition idea. The research context is a text description of the research problem and its background, while the proposition idea presents the potential solution to the problem. In our implementation, we instruct the LLM to generate these research ideas. As LLM, we used gemma2:9B. For simplicity we ask them to use google/gemma-2-2b-it in their study. The prompt used for idea generation is included as Fig. 25 in Appendix. While we opted for a simple prompting approach to idea generation for this study, it's worth noting that there exists a rich body of more sophisticated methods in previous research for research idea generation (Lu et al. 2024; Baek et al. 2024; Yang et al. 2024; Wang et al. 2023; Li et al. 2024a; Pu et al. 2024; Radensky et al. 2024; Gu and Krenn 2024; Hu et al. 2024; Xiong et al. 2024).ncorporating these methods into our framework would improve the quality of generated ideas, while the integration process would be straightforward.

Following idea generation, the script automatically fetch the suitable dataset. Then, the Aider (Aider-AI 2024), an LLM-based code editor previously used in ML research automation (Lu et al. 2024), modifies experiment.py by overriding methods in the NewMLWorkFlow class. This class inherits from MLWorkFlow, which is defined in mlworkflow.py.

The context provided to the code editor included the research idea, information about the automatically acquired dataset and its structure, and details about both the control group (MLWorkFlow) and experimental group (NewMLW orkFlow). The specific tasks given to the editor included implementing method overrides in NewMLWorkFlow and implementing the compare\_and\_evaluate\_proposit ion function that serves as the evaluation method.

After editing experiment.py based on these instructions, run\_controlled\_experiment.py executes that code. The execution process continues up to a predetermined limit, with the code editor receiving any error messages and iteratively refining the code until it runs without errors.

## MLWorkflow

**MLWorkflow Input Structure** Our previous characterization of machine learning as a process requiring (1) a learner and (2) data informs the input structure of MLWor kflow: the class is designed to receive these two components and execute the corresponding processes. The overall process of preparing these two inputs in our current implementation, which will be explained in the following, is depicted in Fig. 4.

For simplicity, we constrained the learner to be goo gle/gemma-2-2b-it in our current implementation. While this is a limitation, it's worth noting that previous research, such as prompt2model (Viswanathan et al. 2023), has demonstrated automatic acquisition of pre-trained models. In fact, prior research has already attempted to automate machine learning research, including model acquisition, by applying these technologies (Li et al. 2024b). By following these previous studies, we could automate even the learner specification process, which would be straightforward to integrate.

For data handling, we implemented an LLM-driven automatic data retrieval system to maintain high abstraction levels in our experimental code templates, making them applicable to diverse research scenarios. Specifically, we implemented a process where an LLM generates appropriate queries for the Huggingface Hub based on the research idea, followed by automatic search, retrieval, and formatting of the data. This enables flexible research code generation that adapts to any changes in research ideas, depending on your research goals.

While current implementation of data retirieval is inspired by prompt2model (Viswanathan et al. 2023), our version is currently simpler for demonstration purposes. Expanding it into a more flexible and sophisticated system, similar to prompt2model, remains future work. The prompts used in this process are shown in Fig. 24 in the Appendix.

MLWorkflow Class Implementation The implementation of MLWorkflow is written in mlworkflow.py, availabe in the GitHub repository. Reflecting our understanding of machine learning as a process of model training followed by execution, MLWorkflow includes two primary methods: train\_model and run\_model. These are called sequentially when the \_\_call\_\_ method is executed. The train\_ model method implements a standard, simple learning procedure that executes training when given training data. The run\_model method implements basic sample code where a trained learner receives test data as input and generates output. Notably, we made train\_model execution optional since not all research interventions require training to validate the effectiveness of their proposition, e.g. new prompt techniques for zero-shot reasoning.

While we currently define only these two methods for simplicity, this broad scope for overriding could be problematic, as the impact of the intervention may extend beyond the intended target. As our understanding and structuring of machine learning processes improve, we could define more granular and appropriate methods, enabling better control over the scope of interventions.



Figure 3: Controlled Experiments as Represented by Inheritance (CERI): Schematic representation of controlled experiments in ML research. As illustrated in Fig. 2, we view the machine learning process (MLWorkflow) as a workflow that takes data and a model as inputs, processing them, and logs intermediate outputs. In CERI framework, introducing a new proposal in machine learning research is understood as an intervention in a MLWorkflow. When representing the original workflow as a base class, this intervention is expressed as an override of methods in a derived class that inherits from the base class. Controlled experiments in ML research are represented as the process of preparing evaluation metrics aligned with the research objectives, comparing outputs from the two workflows, and thereby assessing the validity of the proposition.



Figure 4: Data and Mode Setup Pipeline: In this study, we used gemma-2-2b-it as the model, while data was acquired automatically. First, we prompted the LLM to generate research ideas related to gemma-2-2b-it. Each research idea was structured to include a research context, which describes the research question and background information, and a proposition idea, which outlines the approach to addressing the question. Based on these research ideas, we had the LLM generate queries to search for suitable data on Hugging Face Hub, enabling automatic data acquisition.

# **Experiment Template Code**

The experiment template code named experiment.py contains three key components: (1) the NewMLWorkflow implementation, (2) the evaluation method compare\_and \_evaluate\_proposition, and (3) a main function that executes MLWorkflow and NewMLWorkflow instances and compares them using compare\_and\_evaluate\_p roposition. This file, available in the GitHub repository, serves as the executable script for our controlled experiments and is the target for editing by the coder. **NewMLWorkflow Implementation** NewMLWorkflow is implemented in experiment.py as a derived class that inherits from MLWorkflow, which is imported from a separate file. This structure maintains clear separation between control and experimental implementations. The specific implementation of this function is automatically generated by our code editor as described earlier.

**Evaluation Function Implementation** Following our conceptualization of evaluation methods as functions that process workflow intermediate artifacts to produce evaluation results, we equipped MLWorkflow with a log member variable to store these artifacts. The compare\_and\_evaluate\_proposition function takes these logs as input and outputs evaluation results in a CSV format, with control and experimental groups as rows and evaluation metrics as columns. Again, the specific implementation of this function is automatically generated by our code editor as described earlier.

# Demonstration

## **Quick Look at Prototype System's Functioning**

In the following, we will illustrate how the proposed CERI framework works in practice, with demonstrating an example.

Using the prompts described earlier, we had the system to generate a research idea, full text of which is provided as Figs 26 and 27 in the Appendix. These figures suggest that the LLM identified gemma-2-2b-it's difficulty in solving reasoning tasks as a problem. To address this, it appears to have proposed new prompt engineering techniques, including (1) structured prompting, (2) the incorporation of logical operations, and (3) iterative prompting.

Based on this research idea, we instructed the LLM, using the aforementioned prompt, to retrieve appropriate data. In this current study, we specifically restricted the search to data in the text modality and instructed the system to retrieve the dataset with the highest number of downloads among the results found. As a result, the dataset livebench/reasoning was obtained, which appears to be a dataset designed to measure the logical reasoning capabilities of LLMs. This appears to be an appropriate dataset in light of the objective of analyzing reasoning capabilities. These generated ideas and retrieved dataset serve as the basis for experimental code editing and subsequent execution of the controlled experiment.

Based on the research idea and the retrieved dataset, we had Aider based on claude-3-5-sonnet-20240620 edit the experimental code. The modified code is shown in the Figs. 9 - 12. We confirmed that this code runs without any errors.

Looking at this code, we can see that structural prompting and iterative prompting, as outlined in the research idea, are implemented appropriately. However, the incorporation of logical operators has not yet been fully realized. The implementation of feedback mechanisms also remains limited. Regarding the evaluation metrics, the system assesses whether the generated text includes terms such as "therefore" or "because" (Logical Consistency), which could be considered a simple proxy for everyday "reasoning". On the other hand, since the fetched dataset includes YES/NO answers to questions that require reasoning, it may be more effective to use these targets to evaluate reasoning capabilities. Additionally, the evaluation of the number of steps (Reasoning Steps) seems biased toward metrics that favor the proposer, as the prompt explicitly instructs the LLMs to respond in a "step 1, step 2, ..." format.

The experimental result, saved as a CSV file, is shown in Table 1. As can be seen from the high Logical Consistency (percentage of sentences containing "therefore" or "because"), the proposed method outperformed the baseline in terms of the metrics set by the system in this sense.

ML Workflow	Logical Consistency	Reasoning Steps	Inference Time (s)
Baseline	0.0	0.4333	172.2882
Proposal	0.0111	4.0444	886.7734

Table 1: Comparison of Baseline and Proposal

In summary, there is still significant room for improvement in the quality of the code. A key challenge lies in enhancing the system's ability to identify and appropriately implement suitable evaluation metrics for the proposed method. On the positive side, the approach of isolating and managing the base class to prevent unintended alterations has been successful, and the system has managed to generate error-free, executable code. Despite some quality issues, the automation, including dataset integration and automatic generation of evaluation metrics, shows promising potential for more flexible and autonomous execution of machine learning research.

As mentioned earlier, while there is extensive research on idea generation, the field of flexible hypothesis verification automation, or, experimental code generation is still in its early stages. Building a body of research in this area will be essential for advancing the field.

# **Complexity and Generalizability**

At the beginning, we explained that template code has two key aspects: code generalizability and code complexity. The purpose of this conceptual paper is to propose ideas for designing template code that achieves an optimal balance of generalizability and complexity. Here, we report the findings from a preliminary examination of the proposed template code with respect to each of these aspects.

**Code Generation with vs. without Templates** To discuss the potential of the current template to introduce complexity in the generated code, we conducted a qualitative comparison between code generated without using any template code and code generated using the current template.

As a baseline, we instructed the LLM to generate experimental code without any template guidance. For this comparison, we used claude-3-5-sonnet-20240620, the same model that serves as the base for the Aider, as baseline code generation. The generated code is provided as Figs. 13 - 14 in the Appendix.

The generated baseline code appears well-written. However, because this baseline code was generated based on minimal instructions to simply create experimental code, it only includes a toy dataset rather than a dataset representative of real-world use. In contrast, as previously explained, the template-guided code pre-defines dataset acquisition and integration with machine learning, allowing for the use of more practical datasets.

Furthermore, comparing the length of the code, the baseline generated code is 89 lines, while the template-based code extends to 202 lines, incorporating more complex and detailed processes. This observation suggests that our template code may aid in generating code that resembles the complex code commonly used in human research more closely than when no template code is used. While this experiment has its limitations and does not offer conclusive evidence of this effect, it does provide a tentative indication.

Flexibility in Adapting to Idea Changes To discuss the potential flexibility of the proposed template compared to more domain-specific templates, we compared the code generation results using a domain-specific template with those using the proposed template. Specific templates tailored to narrow research contexts is expected to hinder flexibility when research topics, objectives, or ideas shift. In such cases, the template may fail to function effectively or may even introduce irrelevant noise that negatively impacts the generated code.

Specifically, we used a template code for diffusion model research from a prior study (Lu et al. 2024) as an example baseline and compared the edited code generated using the proposed template code and the baseline code for the previously mentioned dataset, research idea, and model.

The code generated with the specific prompt is presented as Figs. 15 - 22 in the Appendix. The generated code reveals a clear separation between the diffusion-related code segments and the portions implementing the actual methodology. This indicates that the diffusion template is, as expected, entirely non-functional in this context. In summary, it seems that the diffusion experiment template may not have effectively contributed to the current context, aligning with our hypothesis that highly specific template designs have inherent limitations. Although the template's support might be debatable, it hints at the potential limitations associated with overly specialized design approaches.

This stands in contrast to our proposed approach, where the generated code depends on the template code. This observation highlights the possibility that increasing the level of abstraction in template code may enhance flexibility, allowing for broader applicability across different research contexts.

# Conclusion

This conceptual paper introduces an idea of experiment template code design for automating machine learning experiments. Towards a more flexible approach to experimental automation, We propose to represent machine learning process as abstract workflow and controlled experiments as comparisons between base and derived classes reflecting the workflows.

The goal of this conceptual paper is to propose ideas for designing code templates that facilitate the automation of hypothesis validation, particularly in the relatively lessexplored context of validation within the scientific process. Our approach is preliminary and has lots of limitations, and we do not claim it to be the optimal solution. Instead, we hope that this work serves as a catalyst for further innovation, inspiring alternative approaches and improvements in the field.

# References

Aider-AI. 2024. Aider: AI-Powered Assistant for Software Development. Accessed: 2024-11-10.

Baek, J.; Jauhar, S. K.; Cucerzan, S.; and Hwang, S. J. 2024. Researchagent: Iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738*.

Gu, X.; and Krenn, M. 2024. Generation and humanexpert evaluation of interesting research ideas using knowledge graphs and large language models. *arXiv preprint arXiv:2405.17044*.

Hu, X.; Fu, H.; Wang, J.; Wang, Y.; Li, Z.; Xu, R.; Lu, Y.; Jin, Y.; Pan, L.; and Lan, Z. 2024. Nova: An Iterative Planning and Search Approach to Enhance Novelty and Diversity of LLM Generated Ideas. *arXiv preprint arXiv:2410.14255*.

Huang, Q.; Vora, J.; Liang, P.; and Leskovec, J. 2024. MLAgentBench: Evaluating Language Agents on Machine Learning Experimentation. In *Forty-first International Conference on Machine Learning*.

Ifargan, T.; Hafner, L.; Kern, M.; Alcalay, O.; and Kishony, R. 2024. Autonomous LLM-driven research from data to human-verifiable research papers. *arXiv preprint arXiv:2404.17605*.

Li, L.; Xu, W.; Guo, J.; Zhao, R.; Li, X.; Yuan, Y.; Zhang, B.; Jiang, Y.; Xin, Y.; Dang, R.; et al. 2024a. Chain of Ideas:

Revolutionizing Research in Novel Idea Development with LLM Agents. *arXiv preprint arXiv:2410.13185*.

Li, R.; Patel, T.; Wang, Q.; and Du, X. 2024b. Mlr-copilot: Autonomous machine learning research based on large language models agents. *arXiv preprint arXiv:2408.14033*.

Lu, C.; Lu, C.; Lange, R. T.; Foerster, J.; Clune, J.; and Ha, D. 2024. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*.

Pu, K.; Feng, K.; Grossman, T.; Hope, T.; Mishra, B. D.; Latzke, M.; Bragg, J.; Chang, J. C.; and Siangliulue, P. 2024. IdeaSynth: Iterative Research Idea Development Through Evolving and Composing Idea Facets with Literature-Grounded Feedback. *arXiv preprint arXiv:2410.04025*.

Radensky, M.; Shahid, S.; Fok, R.; Siangliulue, P.; Hope, T.; and Weld, D. S. 2024. Scideator: Human-LLM Scientific Idea Generation Grounded in Research-Paper Facet Recombination. *arXiv preprint arXiv:2409.14634*.

Schickore, J. 2022. Scientific Discovery. In Zalta, E. N.; and Nodelman, U., eds., *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2022 edition.

Takagi, S.; Yamauchi, R.; and Kumagai, W. 2023. Towards Autonomous Hypothesis Verification via Language Models with Minimal Guidance. *arXiv preprint arXiv:2311.09706*.

Viswanathan, V.; Zhao, C.; Bertsch, A.; Wu, T.; and Neubig, G. 2023. Prompt2model: Generating deployable models from natural language instructions. *arXiv preprint arXiv:2308.12261*.

Wang, Q.; Downey, D.; Ji, H.; and Hope, T. 2023. Learning to generate novel scientific directions with contextualized literature-based discovery. *arXiv preprint arXiv:2305.14259.* 

Weng, Y.; Zhu, M.; Bao, G.; Zhang, H.; Wang, J.; Zhang, Y.; and Yang, L. 2024. CycleResearcher: Improving Automated Research via Automated Review. *arXiv preprint arXiv:2411.00816*.

Xiong, G.; Xie, E.; Shariatmadari, A. H.; Guo, S.; Bekiranov, S.; and Zhang, A. 2024. Improving Scientific Hypothesis Generation with Knowledge Grounded Large Language Models. *arXiv preprint arXiv:2411.02382*.

Yang, Z.; Du, X.; Li, J.; Zheng, J.; Poria, S.; and Cambria, E. 2024. Large Language Models for Automated Open-domain Scientific Hypotheses Discovery. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 13545–13565. Bangkok, Thailand: Association for Computational Linguistics.

# Appendix

```
from torch.utils.data import DataLoader
from tqdm import tqdm
from transformers import get_linear_schedule_with_warmup
from tqdm import tqdm
def collate_fn(batch):
    input_ids = torch.tensor([item['input_ids'] for item in batch])
    attention_mask = torch.tensor([item['attention_mask'] for item in batch])
    targets = torch.tensor([item['targets'] for item in batch])
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'targets': targets
class MLWorkflow:
    def __init__(self, model, tokenizer, device, tokenize_dataset):
        self.tokenizer = tokenizer
        self.model = model.to(device)
        self.tokenize_dataset = tokenize_dataset
        self.log = {
            "generated_outputs": [],
    def __call__(self, dataset: datasets.Dataset, is_train_included=False):
        if is_train_included:
           self.model, self.tokenizer = self.train_model(dataset["train"])
        log = self.run_model(dataset["test"])
        return log
    def train_model(self, training_datasets: list[datasets.Dataset] | None = None):
        start_time = time.time()
        train_dataset = self.tokenize_dataset(training_datasets, self.tokenizer,
                                              self.tokenizer.model_max_length)
        epochs = 3
        train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True,
        collate_fn=collate_fn)
        optimizer_name = "AdamW"
        optimizer = getattr(optim, optimizer_name)(self.model.parameters(),
                                                   lr=5e-5, weight_decay=0.01)
        total_steps = len(train_loader) * epochs
        scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0,
                                                    num_training_steps=total_steps)
        loss_fn = torch.nn.CrossEntropyLoss()
        for epoch in tqdm(range(epochs), desc="Training"):
```

```
self.model.train()
        total_loss = 0
        for batch in tqdm(train_loader, desc=f"Epoch {epoch + 1}/{epochs}"):
            input_ids = batch['input_ids'].to(self.device)
            attention_mask = batch['attention_mask'].to(self.device)
            targets = batch['targets'].to(self.device)
            outputs = self.model(input_ids, attention_mask=attention_mask, targets=targets)
            logits = outputs.logits
            loss = loss_fn(logits.view(-1, logits.size(-1)), targets.view(-1))
           total_loss += loss.item()
           loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
            optimizer.step()
           scheduler.step()
           optimizer.zero_grad()
        avg_train_loss = total_loss / len(train_loader)
        print(f"Epoch + 1}/{epochs}, Average train loss: {avg_train_loss:.4f}")
        self.log["train_losses"].append(avg_train_loss)
    self.log["train_time"] = time.time() - start_time
    self.log["model_parameters"] = self.model.state_dict()
    self.model.save_pretrained("artifacts")
    self.tokenizer.save_pretrained("artifacts")
    return self.model, self.tokenizer
def run_model(self, test_dataset: datasets.Dataset):
    start_time = time.time()
    test_dataset = self.tokenize_dataset(test_dataset, self.tokenizer,
                                         self.tokenizer.model_max_length)
    test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False,
                            collate_fn=collate_fn)
    all_outputs = []
    self.model.eval()
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(self.device)
            outputs = self.model.generate(
               input_ids,
               attention_mask=batch['attention_mask'],
               max_new_tokens=50,
               num_return_sequences=1,
               do_sample=True,
               pad_token_id=self.tokenizer.pad_token_id
            generated_texts = self.tokenizer.batch_decode(outputs, skip_special_tokens=True)
            all_outputs.extend(generated_texts)
    self.log["inference_time"] = time.time() - start_time
    self.log["generated_outputs"] = all_outputs
```

```
from torch.utils.data import DataLoader
from tqdm import tqdm
from transformers import get_linear_schedule_with_warmup
from mlworkflow import MLWorkflow
def collate_fn(batch):
    input_ids = torch.tensor([item['input_ids'] for item in batch])
    attention_mask = torch.tensor([item['attention_mask'] for item in batch])
    targets = torch.tensor([item['targets'] for item in batch])
       'input_ids': input_ids,
        'attention_mask': attention_mask,
        'targets': targets
class NewMLWorkflow(MLWorkflow):
    def __init__(self, model, tokenizer, device, tokenize_dataset):
       super().__init__(model, tokenizer, device, tokenize_dataset)
def compare_and_evaluate_proposition(log, new_log, test_dataset):
    results = {
    7
        writer.writerow(["mlworkflow", ...])
        for mlworkflow in results:
            writer.writerow([mlworkflow, ...])
def tokenize_dataset(dataset: datasets.Dataset, tokenizer,
                     tokenizer_max_length: int) -> datasets.Dataset:
    def tokenize_function(examples):
        tokenized_inputs = tokenizer(examples[...], truncation=True, padding="max_length",
                                     max_length=tokenizer_max_length)
        tokenized_targets = tokenizer(examples[...], truncation=True, padding="max_length",
                                     max_length=tokenizer_max_length)
        tokenized_inputs["targets"] = tokenized_targets["input_ids"]
        return tokenized_inputs
    tokenized_dataset = dataset.map(tokenize_function, batched=True,
                                    remove_columns=dataset.column_names)
```

return tokenized\_dataset

if \_\_name\_\_ == "\_\_main\_\_": import sys import os sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_))) from model\_preparation import prepare\_model from datasets import load\_from\_disk device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

workspace\_directory = os.path.dirname(os.path.abspath(\_\_file\_\_))
dataset = load\_from\_disk(os.path.join(workspace\_directory, "dataset"))

model, tokenizer = prepare\_model("google/gemma-2-2b-it", is\_pretrained=True)

mlworkflow = MLWorkflow(model, tokenizer, device, tokenize\_dataset)
log = mlworkflow(dataset, is\_train\_included=False)

new\_mlworkflow = NewMLWorkflow(model, tokenizer, device, tokenize\_dataset)
new\_log = new\_mlworkflow(dataset, is\_train\_included=False)

compare\_and\_evaluate\_proposition(log, new\_log, dataset["test"])

print("Finished!!")

```
from torch.utils.data import DataLoader
from tqdm import tqdm
from transformers import get_linear_schedule_with_warmup
from mlworkflow import MLWorkflow
def collate_fn(batch):
    input_ids = torch.tensor([item['input_ids'] for item in batch])
    attention_mask = torch.tensor([item['attention_mask'] for item in batch])
    targets = torch.tensor([item['targets'] for item in batch])
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'targets': targets
class NewMLWorkflow(MLWorkflow):
    def __init__(self, model, tokenizer, device, tokenize_dataset):
        super().__init__(model, tokenizer, device, tokenize_dataset)
        self.structured_prompt_template = """
        Given the following question:
        Please reason step-by-step to answer the question:
        Final answer:
    def run_model(self, test_dataset: datasets.Dataset):
        start_time = time.time()
        all_outputs = []
        self.model.eval()
        with torch.no_grad():
            for item in tqdm(test_dataset, desc="Evaluating"):
                if 'turns' not in item:
                                      {item.keys()}")
                question = item['turns'][0] [0] # Extract the question from the 'turns' column
                    prompt = self.structured_prompt_template.format(
                        question=question,
                        step1="" if i == 0 else all_outputs[-1].split("\n")[0],
                        step2="" if i < 1 else all_outputs[-1].split("\n")[1],</pre>
                        step3="" if i < 2 else all_outputs[-1].split("\n")[2]</pre>
```

```
inputs = self.tokenizer(prompt, return_tensors="pt").to(self.device)
                    outputs = self.model.generate(
                       **inputs,
                       max_new_tokens=100,
                       num_return_sequences=1,
                       do_sample=True,
                       temperature=0.7,
                       pad_token_id=self.tokenizer.pad_token_id
                    )
                    generated_text = self.tokenizer.decode(outputs[0], skip_special_tokens=True)
                    all_outputs.append(generated_text)
        self.log["inference_time"] = time.time() - start_time
        self.log["generated_outputs"] = all_outputs
def compare_and_evaluate_proposition(baseline_log, new_log, test_dataset):
   baseline_outputs = baseline_log["generated_outputs"]
   new_outputs = new_log["generated_outputs"]
   baseline_consistency = evaluate_logical_consistency(baseline_outputs)
   new_consistency = evaluate_logical_consistency(new_outputs)
   baseline_steps = count_reasoning_steps(baseline_outputs)
   new_steps = count_reasoning_steps(new_outputs)
   baseline_time = baseline_log["inference_time"]
   new_time = new_log["inference_time"]
   results = {
            "logical_consistency": baseline_consistency,
            "reasoning_steps": baseline_steps,
           "inference_time": baseline_time
            "logical_consistency": new_consistency,
           "reasoning_steps": new_steps,
            "inference_time": new_time
   print("Evaluation Results:")
   print(f"Baseline Logical Consistency: {baseline_consistency:.4f}")
   print(f"Proposal Logical Consistency: {new_consistency:.4f}")
    if baseline_consistency > 0:
       consistency_improvement = \
            (new_consistency - baseline_consistency) / baseline_consistency * 100
       print(f"Logical Consistency Improvement: {consistency_improvement:.2f}%")
   print(f"Baseline Reasoning Steps: {baseline_steps:.2f}")
   print(f"Proposal Reasoning Steps: {new_steps:.2f}")
    if baseline_steps > 0:
       steps_improvement = (new_steps - baseline_steps) / baseline_steps * 100
       print(f"Reasoning Steps Improvement: {steps_improvement:.2f}%")
```

```
print("Reasoning Steps Improvement: Cannot calculate (baseline is zero)")
    time_change = (new_time - baseline_time) / baseline_time * 100
    print(f"Inference Time Change: {time_change:.2f}%")
        writer = csv.writer(f)
        writer.writerow(["mlworkflow", "logical_consistency", "reasoning_steps", "inference_time"])
        for mlworkflow, data in results.items():
            writer.writerow([mlworkflow, data["logical_consistency"], data["reasoning_steps"],
                              data["inference_time"]])
    return results
def evaluate_logical_consistency(outputs):
    consistent_count = \setminus
       sum(1 for output in outputs if "therefore" in output.lower() and "because" in output.lower())
    return consistent_count / len(outputs)
def count_reasoning_steps(outputs):
    step_counts = [output.count("Step") for output in outputs]
   return sum(step_counts) / len(step_counts)
def tokenize_dataset(dataset: datasets.Dataset, tokenizer,
                     tokenizer_max_length: int) -> datasets.Dataset:
    def tokenize_function(examples):
        # Print the column names and first few entries to debug
print("Dataset columns:", examples.keys())
        for key in examples.keys():
            print(f"First entry of '{key}':", examples[key][0])
        if 'turns' not in examples:
                f"Required 'turns' column not found. Available columns: {examples.keys()}")
        questions = [turn[0] for turn in examples['turns']]
        if 'ground_truth' not in examples:
                f"Required 'ground_truth' column not found. Available columns: {examples.keys()}")
        targets = examples['ground_truth']
        tokenized_inputs = tokenizer(questions, truncation=True, padding="max_length", max_length=tokenizer_ma
        tokenized_targets = tokenizer(targets, truncation=True, padding="max_length", max_length=tokenizer_max
        tokenized_inputs["targets"] = tokenized_targets["input_ids"]
        return tokenized_inputs
    tokenized_dataset = dataset.map(tokenize_function,
                                     batched=True, remove_columns=dataset.column_names)
    return tokenized_dataset
if __name__ == "__main__":
    import sys
```

```
import os
sys.path.append(os.path.dirname(os.path.abspath(__file__)))
from model_preparation import prepare_model
from datasets import load_from_disk
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
workspace_directory = os.path.dirname(os.path.abspath(__file__))
dataset = load_from_disk(os.path.join(workspace_directory, "dataset"))
```

```
# Print dataset info
print("Dataset info:")
print(dataset)
print("Train dataset columns:", dataset["train"].column_names)
print("Test dataset columns:", dataset["test"].column_names)
```

model, tokenizer = prepare\_model("google/gemma-2-2b-it", is\_pretrained=True)

```
mlworkflow = MLWorkflow(model, tokenizer, device, tokenize_dataset)
log = mlworkflow(dataset, is_train_included=False)
```

new\_mlworkflow = NewMLWorkflow(model, tokenizer, device, tokenize\_dataset)
new\_log = new\_mlworkflow(dataset, is\_train\_included=False)

compare\_and\_evaluate\_proposition(log, new\_log, dataset["test"])

print("Finished!!")

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import pandas as pd
from tqdm import tqdm
model_name = "google/gemma-2-2b-it"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
tasks = [
        "premise": "Tutti i gatti sono mammiferi. Whiskers è un gatto.",
         "question": "Whiskers è un mammifero?",
        "question": "Ha piovuto oggi?",
strategies = {
    "baseline": lambda task: f"{task['premise']} {task['question']}",
    "structured": lambda task: f"Premessa: {task['premise']}\nDomanda: \
    {task['question']}\nRisposta:",
"logical_operators": lambda task: f"SE ({task['premise']}) ALLORA \
        {task['question']} (Rispondi Sì o No)",
    "iterative": lambda task: f"Passo 1: Comprendi la premessa: \
        {task['premise']}\nPasso 2: Analizza la domanda: {task['question']}\nPasso 3: \
def generate_response(prompt):
    inputs = tokenizer(prompt, return_tensors="pt")
    with torch.no_grad():
        outputs = model.generate(**inputs, max_new_tokens=50)
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
results = []
for task in tqdm(tasks, desc="Processing tasks"):
    for strategy_name, strategy_func in strategies.items():
        prompt = strategy_func(task)
        response = generate_response(prompt)
        correct = task['answer'].lower() in response.lower()
        results.append({
            "Task": task['question'],
            "Strategy": strategy_name,
            "Prompt": prompt,
             "Response": response,
```

# Convert results to DataFrame and save
df\_results = pd.DataFrame(results)
df\_results.to\_csv("logical\_reasoning\_results.csv", index=False)

# # Calculate and print summary statistics summary = df\_results.groupby("Strategy")["Correct"].mean() print("Accuracy by strategy:") print(summary)

## *# Visualize results*

import matplotlib.pyplot as plt

summary.plot(kind="bar")
plt.title("Accuracy of Logical Reasoning by Prompting Strategy")
plt.ylabel("Accuracy")
plt.tight\_layout()
plt.savefig("strategy\_comparison.png")
plt.close()

#### # Analyze response lengths

df\_results["Response\_Length"] = df\_results["Response"].str.len()
length\_summary = df\_results.groupby("Strategy")["Response\_Length"].mean()

length\_summary.plot(kind="bar")
plt.title("Average Response Length by Prompting Strategy")
plt.ylabel("Characters")
plt.tight\_layout()
plt.savefig("response\_length\_comparison.png")

# This file trains a DDPM diffusion model on 2D datasets.

```
import argparse
import numpy as np
from torch.utils.data import DataLoader
from torch.optim.lr_scheduler import CosineAnnealingLR
from tqdm.auto import tqdm
from transformers import AutoTokenizer, AutoModelForCausalLM
from datasets import load_dataset
import datasets
    print("Warning: npeet not installed. KL divergence estimation will not be available.")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_name = "google/gemma-2-2b-it"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
dataset = load_dataset("livebench/reasoning")
def structured_prompt(question):
Given the following question, let's approach it step-by-step using a decision tree:
Question: {question}
   - [Element 1]
   - [Element 2]
     [Conclusion 1]
     [Conclusion 2]
     [Conclusion 3]
3. Apply logical operators:
```

```
Please fill in the brackets and provide a step-by-step reasoning process.
   return f"""
Let's solve this problem using logical operators:
Question: {question}
  A = [Define A]
  B = [Define B]
  C = [Define C]
3. Interpret the result:
   IF Result is TRUE THEN
    [Conclusion 1]
     [Conclusion 2]
4. Final answer:
Please fill in the brackets and show your reasoning using the logical operators.
def iterative_prompting(question, model, tokenizer, max_iterations=3):
   current_prompt = f"Question: {question}\n\nInitial thoughts:"
    for i in range(max_iterations):
       inputs = tokenizer(current_prompt, return_tensors="pt").to(device)
        outputs = model.generate(**inputs, max_new_tokens=100)
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        current_prompt += f"\n\nIteration {i+1} refinement:\n{response}\n\nCan \
           you improve on this reasoning?"
   return current_prompt
def evaluate_reasoning(model, tokenizer, dataset, num_samples=100):
   correct = 0
   total = 0
    split = next(iter(dataset.keys()))
    first_sample = next(iter(dataset[split]))
    print("Sample structure:", first_sample.keys())
```

```
for sample in tqdm(list(dataset[split])[:num_samples]):
        question = sample.get('input', '') # Assuming 'input' contains the question
ground_truth = sample.get('target', '') # Assuming 'target' contains the ground truth
        structured_input = structured_prompt(question)
        logical_input = logical_operator_prompt(question)
        iterative_input = iterative_prompting(question, model, tokenizer)
        combined_prompt = f"{structured_input}\n\n{logical_input}\n\n{iterative_input}\n\nFinal \
        inputs = tokenizer(combined_prompt, return_tensors="pt").to(device)
        outputs = model.generate(**inputs, max_new_tokens=50)
        predicted_answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
        if str(ground_truth) in predicted_answer:
        total += 1
    accuracy = correct / total
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--num_samples", type=int, default=100)
    parser.add_argument("--out_dir", type=str, default="run_0")
    config = parser.parse_args()
    pathlib.Path(config.out_dir).mkdir(parents=True, exist_ok=True)
    accuracy = evaluate_reasoning(model, tokenizer, dataset, num_samples=config_num_samples)
    results = {
        "accuracy": accuracy,
        "num_samples": config.num_samples,
    7
    with open(osp.join(config.out_dir, "results.json"), "w") as f:
        json.dump(results, f)
    print(f"Evaluation complete. Accuracy: {accuracy:.2f}")
class SinusoidalEmbedding(nn.Module):
        self.dim = dim
    def forward(self, x: torch.Tensor):
        half_dim = self.dim // 2
        emb = torch.log(torch.Tensor([10000.0])) / (half_dim - 1)
        emb = torch.exp(-emb * torch.arange(half_dim)).to(device)
        emb = x.unsqueeze(-1) * emb.unsqueeze(0)
        emb = torch.cat((torch.sin(emb), torch.cos(emb)), dim=-1)
```

Figure 17: Generated Code with Diffusion Experiment Template (3/8)

```
class ResidualBlock(nn.Module):
   def __init__(self, width: int):
       self.ff = nn.Linear(width, width)
       self.act = nn.ReLU()
   def forward(self, x: torch.Tensor):
       return x + self.ff(self.act(x))
class MLPDenoiser(nn.Module):
           self.
            embedding_dim: int = 128,
            hidden_dim: int = 256,
            hidden_layers: int = 3,
       self.time_mlp = SinusoidalEmbedding(embedding_dim)
       self.input_mlp1 = SinusoidalEmbedding(embedding_dim, scale=25.0)
       self.input_mlp2 = SinusoidalEmbedding(embedding_dim, scale=25.0)
       self.network = nn.Sequential(
            nn.Linear(embedding_dim * 3, hidden_dim),
            *[ResidualBlock(hidden_dim) for _ in range(hidden_layers)],
            nn.ReLU(),
            nn.Linear(hidden_dim, 2),
       x1_emb = self.input_mlp1(x[:, 0])
       x2_emb = self.input_mlp2(x[:, 1])
       t_emb = self.time_mlp(t)
       emb = torch.cat([x1_emb, x2_emb, t_emb], dim=-1)
       return self.network(emb)
class NoiseScheduler():
            num_timesteps=1000,
            beta_start=0.0001,
            beta_end=0.02,
            beta_schedule="linear",
        self.num_timesteps = num_timesteps
       if beta_schedule == "linear":
            self.betas = torch.linspace(
               beta_start, beta_end, num_timesteps, dtype=torch.float32).to(device)
       elif beta_schedule == "quadratic":
            self.betas = (torch.linspace(
               beta_start ** 0.5, beta_end ** 0.5,
               num_timesteps, dtype=torch.float32) ** 2).to(device)
            raise ValueError(f"Unknown beta schedule: {beta_schedule}")
       self.alphas = 1.0 - self.betas
```

return emb

Figure 18: Generated Code with Diffusion Experiment Template (4/8)

```
self.alphas_cumprod = torch.cumprod(self.alphas, axis=0).to(device)
    self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1], (1, 0), value=1.).to(device)
    self.sqrt_alphas_cumprod = (self.alphas_cumprod ** 0.5).to(device)
    self.sqrt_one_minus_alphas_cumprod = ((1 - self.alphas_cumprod) ** 0.5).to(device)
    self.sqrt_inv_alphas_cumprod = torch.sqrt(1 / self.alphas_cumprod).to(device)
    self.sqrt_inv_alphas_cumprod_minus_one = torch.sqrt(
        1 / self.alphas_cumprod - 1).to(device)
    self.posterior_mean_coef1 = \
        self.betas * torch.sqrt(self.alphas_cumprod_prev) / (1. - self.alphas_cumprod).to(
        device)
    self.posterior_mean_coef2 = ((1. - self.alphas_cumprod_prev) * torch_sqrt(self.alphas) / (
           1. - self.alphas_cumprod)).to(device)
def reconstruct_x0(self, x_t, t, noise):
    s1 = self.sqrt_inv_alphas_cumprod[t]
    s2 = self.sqrt_inv_alphas_cumprod_minus_one[t]
   s1 = s1.reshape(-1, 1)
   s2 = s2.reshape(-1, 1)
def q_posterior(self, x_0, x_t, t):
    s1 = self.posterior_mean_coef1[t]
    s2 = self.posterior_mean_coef2[t]
    s1 = s1.reshape(-1, 1)
    s2 = s2.reshape(-1, 1)
   mu = s1 * x_0 + s2 * x_t
   return mu
       return 0
    variance = self.betas[t] * \
       (1. - self.alphas_cumprod_prev[t]) / (1. - self.alphas_cumprod[t])
    variance = variance.clip(1e-20)
   return variance
def step(self, model_output, timestep, sample):
    t = timestep
    pred_original_sample = self.reconstruct_x0(sample, t, model_output)
    pred_prev_sample = self.q_posterior(pred_original_sample, sample, t)
        noise = torch.randn_like(model_output)
        variance = (self.get_variance(t) ** 0.5) * noise
    pred_prev_sample = pred_prev_sample + variance
    return pred_prev_sample
def add_noise(self, x_start, x_noise, timesteps):
    s1 = self.sqrt_alphas_cumprod[timesteps]
    s2 = self.sqrt_one_minus_alphas_cumprod[timesteps]
```

```
s1 = s1.reshape(-1, 1)
       s2 = s2.reshape(-1, 1)
       return s1 * x_start + s2 * x_noise
        return self.num_timesteps
if __name__ == "__main__":
   parser = argparse.ArgumentParser()
   parser.add_argument("--train_batch_size", type=int, default=256)
   parser.add_argument("--eval_batch_size", type=int, default=10000)
   parser.add_argument("--learning_rate", type=float, default=3e-4)
   parser.add_argument("--num_timesteps", type=int, default=100)
    parser.add_argument("--num_train_steps", type=int, default=10000)
   parser.add_argument("--beta_schedule", type=str, default="linear",
                        choices=["linear", "quadratic"])
   parser.add_argument("--embedding_dim", type=int, default=128)
   parser.add_argument("--hidden_size", type=int, default=256)
   parser.add_argument("--hidden_layers", type=int, default=3)
   parser.add_argument("--out_dir", type=str, default="run_0")
   config = parser.parse_args()
   final_infos = {}
   all_results = {}
   pathlib.Path(config.out_dir).mkdir(parents=True, exist_ok=True)
    for dataset_name in ["circle", "dino", "line", "moons"]:
       dataset = getattr(datasets, dataset_name)(n=100000)
       dataloader = DataLoader(dataset, batch_size=config.train_batch_size, shuffle=True)
       model = MLPDenoiser(
            embedding_dim=config.embedding_dim,
            hidden_dim=config.hidden_size,
            hidden_layers=config.hidden_layers,
        ).to(device)
       noise_scheduler = NoiseScheduler(num_timesteps=config.num_timesteps,
                                         beta_schedule=config.beta_schedule)
       optimizer = torch.optim.AdamW(
            model.parameters(),
            lr=config.learning_rate,
       scheduler = CosineAnnealingLR(optimizer, T_max=config num_train_steps)
        train_losses = []
       model.train()
       global_step = 0
       progress_bar = tqdm(total=config.num_train_steps, mininterval=10, disable=True)
       progress_bar.set_description("Training")
       start_time = time.time()
       while global_step < config.num_train_steps:</pre>
            for batch in dataloader:
                if global_step >= config.num_train_steps:
```

```
batch = batch[0].to(device)
       noise = torch.randn(batch.shape).to(device)
        timesteps = torch.randint(
            0, noise_scheduler_num_timesteps, (batch_shape[0],)
        ).long().to(device)
       noisy = noise_scheduler.add_noise(batch, noise, timesteps)
        noise_pred = model(noisy, timesteps)
        loss = F.mse_loss(noise_pred, noise)
       loss.backward()
       nn.utils.clip_grad_norm_(model.parameters(), 0.5)
       optimizer.step()
       optimizer.zero_grad()
       scheduler.step()
       progress_bar.update(1)
        logs = {"loss": loss.detach().item()}
        train_losses append(loss detach() item())
       progress_bar.set_postfix(**logs)
       global_step += 1
progress_bar.close()
end_time = time.time()
training_time = end_time - start_time
model.eval()
eval_losses = []
for batch in dataloader:
   batch = batch[0].to(device)
   noise = torch.randn(batch.shape).to(device)
   timesteps = torch.randint(
       0, noise_scheduler_num_timesteps, (batch_shape[0],)
    ).long().to(device)
    noisy = noise_scheduler.add_noise(batch, noise, timesteps)
    noise_pred = model(noisy, timesteps)
    loss = F.mse_loss(noise_pred, noise)
    eval_losses.append(loss.detach().item())
eval_loss = np.mean(eval_losses)
model.eval()
sample = torch.randn(config.eval_batch_size, 2).to(device)
timesteps = list(range(len(noise_scheduler)))[::-1]
inference_start_time = time.time()
for t in timesteps:
    t = torch.from_numpy(np.repeat(t, config.eval_batch_size)).long().to(device)
    with torch.no_grad():
       residual = model(sample, t)
    sample = noise_scheduler.step(residual, t[0], sample)
sample = sample.cpu().numpy()
inference_end_time = time.time()
inference_time = inference_end_time - inference_start_time
real_data = dataset.tensors[0].numpy()
kl_divergence = ee.kldiv(real_data, sample, k=5) if ee is not None else None
final_infos[dataset_name] = {
```

```
"means": {
    "training_time": training_time,
    "eval_loss": eval_loss,
    "inference_time": inference_time,
    "kl_divergence": inference_time,
    "kl_divergence": kl_divergence,
    }
}
all_results[dataset_name] = {
    "train_losses": train_losses,
    "images": sample,
}
with open(osp.join(config.out_dir, "final_info.json"), "w") as f:
    json.dump(final_infos, f)
```

```
with open(osp.join(config.out_dir, "all_results.pkl"), "wb") as f:
    pickle.dump(all_results, f)
```



Research Context: {research\_context} Proposition Idea: {proposition\_idea}

Prerequisite: Model: The base model used in this experiment is google/gemma-2-2b-it.

Dataset: {dataset\_name} {dataset}

mlworkflow.py: {mlworkflow\_py}

#### Code Explanation:

mlworkflow.py represents the workflow of a machine learning research that verifies the effectiveness of the proposed method through comparative experiments. Specifically, given the dataset, model, and tokenizer, it executes MLWorkflow and NewMLWorkflow (which is a modified version of MLWorkflow), and then compares and evaluates their results using compare\_and\_evaluate\_proposition

MLWorkflow represents a typical machine learning workflow where the model is trained on training data and then executed on test data.

experiment.py represents the comparative experiment that validates the effectiveness of the proposed method through the comparison between MLWorkflow and NewMLWorkflow.

NewMLWorkflow inherits from MLWorkflow and modifies the workflow by overriding train\_model, run\_model, or both. Researcher implements the proposed idea in NewMLWorkflow and the proposition idea and related parts are the only differences between MLWorkflow and NewMLWorkflow.

This code embodies the idea that "machine learning research that validates a proposal through comparative experiments is an endeavor to determine whether adding a new proposal (NewMLWorkflow) to an MLWorkflow that generates certain output from data yields better results in an expected sense."

#### Task Description:

Please edit experiment.py to implement the Proposition Idea and design experiments to validate its feasibility based on the Research Context.

Your task is to complete the experimental code by editing experiment.py. Please edit the following parts, but do not change any other parts:

### NewMLWorkflow

To implement the Proposition Idea and design experiments to validate its effectiveness, override one or all methods of MLWorkflow. For example, if you're proposing a new Optimizer, implement the new optimizer and use it in train\_model instead of the existing optimizer. If you're proposing a new neural architecture, implement the new architecture in the Hugging Face format and assign it in the part where self.model = model.to(device) is set in the \_\_init\_\_ method. If you're proposing a prompt technique to improve the zero-shot inference performance of a pre-trained model, implement the prompt technique in the run\_model part. In this way, first consider which part of the machine learning workflow the Proposition Idea is addressing, and then implement NewMLWorkflow to properly implement and experiment with the proposal. When doing so, make sure to define all the information needed to see if the proposal is superior to existing methods in the expected sense using self.log.

#### compare\_and\_evaluate\_proposition

Implement evaluation criteria to examine how and in what sense the Proposition Idea is superior to existing methods. For example, if the proposed method is expected to predict better than existing methods, you might compare if the accuracy is higher. Or, if you're proposing an optimization method that's expected to converge faster, you might compare the number of steps it took before the loss reached a certain value. Also, if you're proposing a method with superior interpretability, you might define some metric that shows that the internal representation of the model is more interpretable in some sense and compare that. In this way, consider in what sense the Proposition Idea is expected to be superior to existing methods in relation to the Research Context, and implement evaluation metrics that can compare this.

#### tokenize\_dataset

Implement the tokenize\_dataset function to convert the dataset into a format suitable for the current research context. Input and target should be chosen appropriately based on the current research context and dataset structure. Research Context: {research\_context} Proposition\_Idea: {proposition\_idea}

Your final goal is to implement a Proposition Idea and design experiments to validate its feasibility based on the Research Context. To achieve this, generate search query for obtaining text datasets from HuggingFace using an LLM. This prompt should explicitly state the purpose and objectives of the research, what the Proposition Idea is, and which datasets should be retrieved to validate it.

query will be used as follows: from huggingface\_hub import HfApi api = HfApi() results = list(api.list\_datasets(search=query, sort="downloads", direction=-1, limit=max\_results))

Note that query input to `search` is a string that will be contained in the returned datasets.

The query should be a word that is likely included in the desired dataset name but unlikely to be found in unrelated dataset names.

It should primarily consist of terms related to machine learning tasks or concepts. In particular, it should include words associated with the operations required for validating the method you proposed with the dataset or task.

First, think about the machine learning \*\*task\*\* that to be used in the experiment. Then, generate a query that is likely to be included in the dataset name to be used for the task.

Generate only one query and the query should be single word contined in the text dataset name.

<query> "..." </query>

Figure 24: Prompt for retrieving dataset from Huggingface Hub

There is a powerful yet small-scale local LLM called google/gemma-2-2b-it. You will now conduct research using this model.

The research could propose optimization methods, new neural architectures, new evaluation methods, or new prompting techniques, or any other ideas.

Please generate the following two items:

research\_context: This describes the research problem and its background.

proposition\_idea: This describes the proposed idea to solve the problem. Please focus on the single most interesting idea.

Please keep the proposed ideas as simple as possible and ensure they have a high likelihood of feasibility.

<research\_context> ... </research\_context> <proposition\_idea> ... </proposition\_idea>

# Figure 25: Prompt for Generating Research Idea

google/gemma-2-2b-it is a powerful yet small-scale LLM, specifically designed for Italian language tasks. While its performance is impressive for its size, LLMs in general often struggle with reasoning and logical deduction tasks. This can be attributed to their reliance on statistical patterns in the training data rather than explicit logical rules. Furthermore, evaluating the performance of smaller LLMs like gemma-2-2b-it on complex reasoning tasks can be challenging due to the lack of widely accepted benchmarks tailored for their scale.

Figure 26: Research context generated by LLM

To address these challenges, we propose exploring \*\*prompt engineering techniques specifically designed to guide gemma-2-2b-it towards better logical reasoning\*\*. This could involve: \* \*\*Introducing structured input formats\*\*: Instead of relying solely on free-form text, experiment with prompting gemma-2-2b-it using structured data like knowledge graphs or decision trees. \* \*\*Incorporating logical operators\*\*: Train gemma-2-2b-it to understand and utilize logical operators (AND, OR, NOT) within prompts to guide its reasoning process. \* \*\*Utilizing iterative prompting\*\*: Develop a sequence of prompts that gradually lead gemma-2-2b-it towards the desired solution, incorporating intermediate steps and feedback mechanisms. This approach focuses on leveraging gemma-2-2b-it's existing strengths while mitigating its weaknesses in logical reasoning through targeted prompt engineering. It is relatively simple to implement and has a high likelihood of feasibility given the existing research on prompt engineering for LLMs.

Figure 27: Proposition idea generated by LLM