DL-Bench: Code Generation Benchmark Dataset for Deep Learning

Anonymous Authors

Abstract—

I. Abstract

Deep learning (DL) has revolutionized various fields, enabling significant advancements in areas such as computer vision, natural language processing, and more. However, many DL systems are developed by domain experts rather than software developers, creating coding challenges due to the complexity of DL workflows. Large Language Models (LLMs), such as GPT-4o, have emerged as promising tools to assist in DL code generation, offering potential solutions to these challenges. Despite this, current benchmarks, such as DS-1000, are limited and focus primarily on small DL code snippets related to pre/post-processing tasks and lack comprehensive coverage across the full DL pipeline, includes different DL phases and input types.

To address this, we introduce DL-Bench, a novel benchmark dataset designed for functional-level DL code generation. DL-Bench categorizes DL phases (e.g., pre-processing, model construction, training), tasks (e.g., classification, regression, recommendation), and input types (e.g., tabular, image, text). When applied to DL-Bench, GPT-4o showed an overall accuracy of 31% which is significantly lower than the existing dataset DS-1000 accuracy of 51% which indicates the higher difficulty of DL-Bench. There are also notable differences in performance across categories which can be as much as 7% among phases and 39% among tasks. These differences suggest that DL-Bench provides useful insights that can help improve LLMs' performance.

Overall, our empirical results demonstrate the utility of DL-Bench as a comprehensive benchmark while offering insights for future improvements across diverse functional categories.

II. Introduction

In recent years, machine learning (ML) and deep learning (DL) have advanced significantly and have been integrated into various fields [1], [2], [3]. DL coding has its challenges [4] and because of its widespread use, many DL systems are developed by domain experts who are often not software developers [5], [6], which compounds the problems even more.

Recently with the rise of Large Language Models (LLMs) such as ChatGPT, LLMs are considered one of the best solutions to aid with coding [7], [8], [9]. As shown in Table I, There are many code generation datasets [10], [11], [12], [13], [14], [15], [16] which can be used to benchmark LLMs' code generation capability. However, there is only DS-1000 [17] that provides some benchmarks for a limited set of DL code generation categories. Specifically, they provide only entries with small scripts like DL code snippets (typically a couple of lines) which perform mostly on the pre/post-processing. Furthermore, they do not include categorizations such as DL phases (e.g., pre/post-processing, model construction, training, inference, and evaluation) or input types (e.g., tabular, image, or text) which can provide valuable insight for future DL code generation research.

To address this gap, we introduce DL-Bench, a novel dataset that provides benchmarking[18] for DL code generation at a functional level. It includes the code generation prompt, the ground-truth code at the function level, an extensive set of unit tests, and three categorizations (DL phases, ML tasks, and input types). Unlike DS-1000, DL-Bench provides a more comprehensive set of entries that represent the full pipeline of DL system, contains code examples for various machine learning tasks, and collects DL functions with a wide range of inputs ranging from tabular, image, and text. Specifically, each entry DL-Bench dataset is categorized based on these three categories:

- The DL/ML pipeline stages [19], [20] (consists of pre/post-processing, model construction, training, inference, and evaluation).
- The DL/ML tasks [21], [22] (consists of classification, object detection, image segmentation, time-series prediction, recommendation, and regression).
- The input data types [23] (consists of text, image, and array).

These categorizations provide an opportunity for a more in-depth evaluation of future DL code generation techniques.

To demonstrate how useful DL-Bench can be in evaluating DL code generation approaches, we applied GPT-4o [9], OpenAI's latest LLM, on DL-Bench and analyze in-depth its DL code generation capability. Our study found that while GPT-4o showed strong performance in various code generation tasks, it achieved an accuracy of only 31% on DL-Bench. This accuracy is significantly lower than the reported accuracy of 51% for the existing dataset DS-1000, suggesting that DL-Bench contains more challenging data points. Furthermore, the difficulty of generating code varies significantly across categories. For example, GPT-4o reaches an accuracy of 33% for pre/post-processing tasks but only 26% for model construction. Additionally, the accuracies vary even more among task types, ranging from 58% for recommendation tasks to 19% for segmentation tasks. These large gaps in performance between categories demonstrate the insights that DL-Bench can bring to help improve LLM DL code generation capability.

This paper makes the following contributions:

- DL-Bench, a novel benchmark dataset for DL code generation which contains:
  - High quality and diverse source code at the function level collected from GitHub

TABLE I: Comparison of code generation benchmarks. In the "Level" column, "F" represents functions and "S" represents statements. In the "Source" column, "MC" means manually curated, "PC" indicates programming competitions, "GH" refers to GitHub, and 'SO' refers to Stack Overflow.

Benchmark	**Size**	**Language**	**Level**	**Source**	**Metrics**
Human Eval[24]	164	Python	F	MC	Pass@k
MBPP[11]	974	Python	F	MC	Pass@k
APPS[10]	10000	Python	F	PC	Pass@k
CoderEval[25]	460	Python-Java	F	GH	Pass@k
RepoEval[26]	1600	Python	F	GH	EM,ES
DS-1000[17]	1000	Python	S	SO	Pass@k
DL-Bench	**520**	**Python**	**F**	**GH**	**Pass@k**

  – Three types of categorization: DL pipeline stages, data types, and ML task types
- A study demonstrating DL-Bench capability to perform an in-depth evaluation of the state-of-the-art LLMs such as GPT-4o, Claude3.5 sonet, LLama 3.1 70B, and Mistral 7B.

DL-Bench's data is available in our public repository[1]

III. Benchmark Construction

In this section, we will discuss the step-by-step process of constructing DL-Bench as demonstrated in Figure 1. The construction process of DL-Bench consists of two main phases: the Raw Data Extraction and the Labeling Procedure. The raw data extraction involves six semi-automatic steps. Since DL-Bench is designed to have diverse and realistic code samples, the first step ① is to construct DL-Bench from code crawled from highly rated GitHub repositories (i.e., with the most stars) filtered using 30 DL-related terms such as "neural-networks", "pytorch", "computer-vision". We then manually select (step ②) 160 high quality candidate DL projects (i.e., involve the integration of DL and AI-related frameworks, comprehensive test cases, clear and well-written docstrings, and detailed contribution guidelines). We then employed a bespoke utility to extract the test files and then test cases from each repository (step ③ and ④). By performing static analysis, we were able to track and collect all of the functions under test in step ⑤ to form the raw data that is the base of DL-Bench.

Once the raw data is extracted, the labeling procedure starts. To speed up the task of constructing the prompt for each code sample, we utilize LLM (i.e., GPT-4o) as a code-explanation tool[27] to generate the first prompt candidate for each function under test (step ⑦). Four co-authors were then tasked with manually filtering each entry to ensure that each function is highly relevant (i.e., contributes to a DL task such as image recognition, utilizes at least one recognized DL framework, and implements a relatively advanced and sophisticated algorithm). Finally, we conduct a manual labeling process involving four co-authors (step ⑨) to refine the prompt and label each code sample with the appropriate category from our three chosen types of categories: DL pipeline phases, ML task types, and input types.

[1]...

A. Raw Data Extraction

This phase consists of six semi-automatic steps that crawl data from GitHub repositories to generate a list of function definitions and their test cases.

1) Repository Selection: One of the characteristics of DL-Bench is the source of its origin. Unlike other datasets, which may draw from limited or outdated sources, we have curated our data from the most starred DL-related GitHub repositories [28], [29]. This ensures that our dataset contains high-quality and widely used functions that reflect current trends and best practices in DL. To do this, we follow two steps.

In step ①, we filtered GitHub projects with one of 30 DL-related tags such as "neural-networks", "pytorch", and "computer-vision" (we provided the complete list of tags in our repository). To select the most rated project, we select our candidate to be in the top 1000 projects ranked by the number of stars. Using this amount of tags ensures that the tags remain highly relevant and focused on popular, well-defined DL subdomains. Specifically, we select the tags by collecting from DL and AI-related GitHub repositories and filtering the most relevant ones to get the final 30.

In step ②, we manually select the most relevant projects for DL-Bench. Specifically, from the 1000 candidates, we review each project and retain only those that: 1) are DL related (i.e., use DL libraries like TensorFlow or PyTorch, or perform tasks like segmentation or detection), 2) have sufficient test cases (averaging at least three per function), and 3) include thorough documentation, such as source code docstrings or README files. This ensures our selected repositories are relevant, well-documented, and actively maintained, providing reliable ground-truth code for our benchmark [30]. In the end, we select 160 projects, focusing on repositories updated after GPT-4o release to minimize data leakage.

2) Function Extraction: One of the main design choices of DL-Bench is to include a set of reliable and robust test cases for each benchmark entry. This is because programming languages are different from natural languages. Specifically, generated code can fulfill all of the functional requirements but could have a low BLEU score when compared with the ground truth code[31]. This means that using text similarity metrics such as BLEU score as evaluation metrics is not the best method to evaluate code generation techniques. Instead, test cases (functional and non-functional) passing rate should be used to reliably access a new code generation approach.

In step ③, we crawled selected repositories for test files using standard test file name patterns such as tests/test_file_name.py [32]. In step ④, for each test file, we extract test cases using common patterns in Python test suites, such as the @pytest decorator.

Once we identified all test cases, in step ⑤, we performed call graph analysis to track and collect all functions under test (excluding third-party function calls). The definitions of each

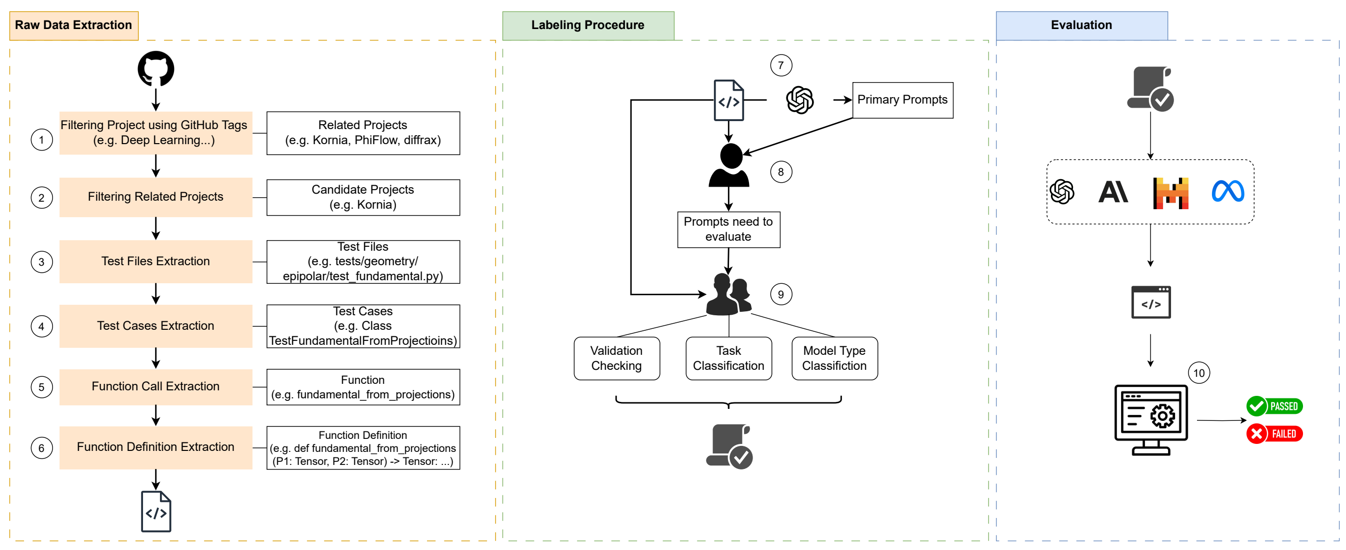

Raw Data Extraction
Labeling Procedure
Evaluation
1
Filtering Project using GitHub Tags (e.g. Deep Learning...)
Related Projects (e.g. Kornia, PhiFlow, diffrax)
2
Filtering Related Projects
Candidate Projects (e.g. Kornia)
3
Test Files Extraction
Test Files (e.g. tests/geometry/epipolar/test_fundamental.py)
4
Test Cases Extraction
Test Cases (e.g. Class TestFundamentalFromProjectioins)
5
Function Call Extraction
Function (e.g. fundamental_from_projections)
6
Function Definition Extraction
Function Definition (e.g. def fundamental_from_projections (P1: Tensor, P2: Tensor) -> Tensor: ...)
</>
7
Primary Prompts
8
Prompts need to evaluate
9
Validation Checking
Task Classification
Model Type Classification
10
PASSED
FAILED

Fig. 1: DL-Bench construction procedure

Create a prompt using the code and its **doc-string**, including the **function/class** name, **inputs**, and **outputs**.

Fig. 2: Template of generating prompt from code

of those functions are then extracted in step 6 to form the bases for our ground-truth code samples.

B. Labeling Procedure

The labeling procedure involves three semi-automatic steps to generate and refine a prompt and assign categorizations for each entry in our DL-Bench dataset. To determine the best procedure and criteria for our manual process, we perform a small trial run of the manual process on a small sample of the data points. In this trial run, we ask each reviewer to provide feedback on the labeling criteria so that when we start our full run we have the most comprehensive and accurate manual process possible.

1) Prompt Generation: Since the DL-Bench is designed to be a benchmark for code generation with LLM, one of the most important components is the prompts associated with the collected source code. In step 7, we utilize two sources of data to create such prompts: 1) the doc-strings provided by developers, which describe the functionality and parameters of the code, and 2) the function definitions themselves, which can be used to generate candidate prompts. Specifically, We take advantage of the function definitions to explain the code, and by combining them with their respective doc-strings (when available), we generate the initial candidate prompt by querying GPT-4o with the template as described in Fig. 2.

However, generated prompts require manual checking and validation to ensure their accuracy and relevancy. This manual review process is essential to refine the prompts and guarantee their quality for subsequent use [33]. We further refine the prompt by considering the following questions:

Does the prompt contain clear sufficient information for the code to be generated? This assessment aims to ensure the prompt's clarity and comprehensibility for a human expert. Annotators check that the prompt includes all essential variables, functions, and definitions for high-quality code generation, providing enough information to clearly explain the problem. The human expert serves as the benchmark to set a high standard for future code generation. We also verify that the prompt provides sufficient guidance, including specific coding conventions or required components.

Does the prompt specify the input and output format? Since our test cases require certain input and output formats, it is important to check such details in the candidate prompt to enable our test cases to function correctly[34], [35]. In other words, without precise definitions of the input and output specifications, the generated code might not align with the expected test parameters, resulting in false negative results during evaluation. Error and exception handling are also considered in this question. For example, we specifically check whether the prompt accounts for handling cases such as "ValueError", "TypeError", or other domain-specific exceptions that the function might raise. This will ensure that the code will be correctly evaluated given our extracted test cases.

Does the prompt cover error handling and boundary conditions? Similar to input and output specification, error handling and boundary conditions are often part of the required testing parameters By ensuring that the prompt includes such details, we ensure that the passing rate truly reflects the performance of the code generation under test.

If the prompt does not meet mentioned criteria, the annotators propose and agree on changes that bring it up to the expected quality. This reviewing process produce prompts that are not only technically correct but also include details essential to code generation.

2) Data Filtering and Validation: After compiling all data (i.e., the ground truth, test cases, and candidate prompts), in step ⑧, we manually evaluate each function meticulously, reading and modifying the prompts following a set of criteria. Specifically, we discard general codes (e.g., those for reading text files) that are not DL related. In this step, the annotators independently assess the prompt's clarity, relevance to DL-related tasks, and overall usability with the following criteria:

- **Serving key DL tasks:** The prompt and the associated function should be closely aligned with significant DL tasks such as image recognition, regression, item recommendation, object detection, label prediction, and natural language processing tasks. This criterion ensures that our dataset contains all important and relevant data points[36].
- **Utilization of popular DL frameworks:** The code should efficiently use widely recognized AI frameworks (when appropriate), such as TensorFlow, PyTorch, or Keras. This criterion ensures our dataset represents typical DL code with a heavy emphasis on reusability[37].
- **Algorithms' relevancy and clarity:** The code should implement DL-specific algorithms (e.g., edge detection algorithms, Principal component analysis, or Stochastic gradient descent). The code should also be well-documented and easy to understand. Complex algorithms must strike a balance between technical depth and clarity to ensure usability.

3) Labeling: In step ⑨, we assign labels for each data point based on the role of the function in the ML pipeline (e.g., pre/post-processing, model construction), the ML tasks (e.g., classification, regression) it solves, and types of data (e.g., image, text) it operates on. For each data point, three co-authors thoroughly analyze and assign appropriate labels. We use a majority vote to finalize the labels and modify the prompts accordingly. Specifically, we assign the following labels when appropriate to each data point: Stage in the ML pipeline, ML task type, and Input data type.

Stage in the ML pipeline: This label indicates the stage that the code is in within the ML pipeline: *Pre/post Processing*, *Model Construction*, *Training*, *Inference*, or *Evaluation & Metrics*. The annotators determine whether the function is related to a stage by analyzing the code and comment to find information that is related to the specific stage. For example, code that specifies a convolutional neural network (CNN) architecture with layers such as convolutions or pooling would fall under the Model Construction category.

ML task type: This label indicates the ML task[38], [39], [40] that the code is serving when applicable. The annotators examine the code to determine the type of task being solved, such as *Time series Prediction*, *Recommendation*, *Image Segmentation*, *Object Detection*, *Regression*, *Classification*, or *General*. Specifically, the annotators look for patterns in the code corresponding to each task. For instance, code that outputs bounding boxes and class labels for objects falls under the Object Detection category. In cases where the code can be used for multiple ML tasks (i.e., does not exclusively belong to a specific ML task), we assigned a *General* label.

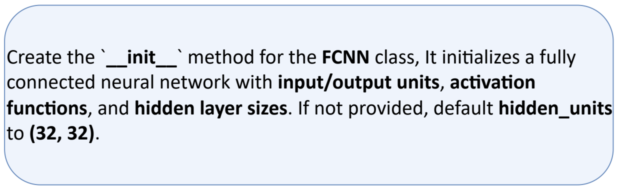

Create the `__init__` method for the **FCNN** class, It initializes a fully connected neural network with **input/output units**, **activation functions**, and **hidden layer sizes**. If not provided, default **hidden_units** to **(32, 32)**.

Fig. 3: An example prompt for Model Construction

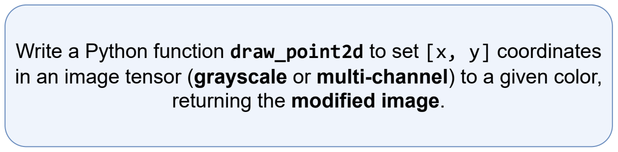

Write a Python function **`draw_point2d`** to set `[x, y]` coordinates in an image tensor (**grayscale** or **multi-channel**) to a given color, returning the **modified image**.

Fig. 4: An example prompt for Pre/Post processing

Input data type: This label indicates the input data type of the function. We focus on typical ML input data types such as *Image*, *Text*, *Structured Array* (i.e., tabular), and *Others*. The annotators analyze the processing flow of data to assign accurate labels. For example, techniques like flipping, cropping, or adding noise process image input. When the input data does not fit one of the typical types (image, text, structured array), we assign the Others label. More details for each category are provided in our repository[2].

Once each reviewer completes their assessments, the team meets to discuss any discrepancies and reach a consensus on the final labels. Due to our detailed instructions and guidelines, we achieve high inter-rater reliability of xx measured by Krippendorff's alpha [41](measures of more than 0.8 indicating strong agreement.

The labeled data is carefully documented, including notes on the decision-making process for transparency and future reference. Instances are organized, with labels to ensure easy retrieval and analysis in later stages of research. To enable easier benchmark utilization (i.e., running test cases), the relevant projects are set up in virtual environments along with appropriate dependencies and ready-to-run testing scripts.

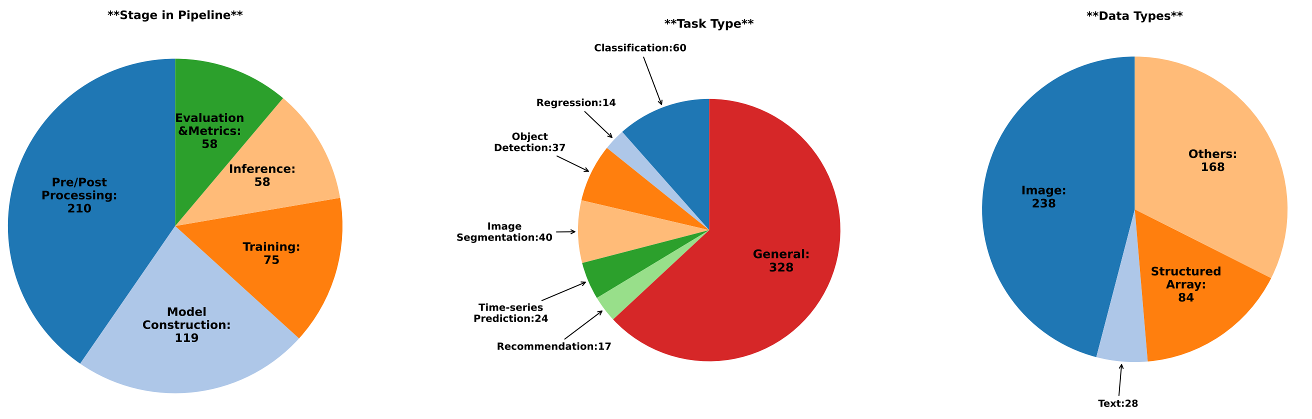
This rigorous review and labeling process ensures that each instance in the dataset is not only relevant and useful but also thoroughly understood and appropriately categorized, contributing to a robust and reliable benchmark.

IV. Statistics of Dataset

In this section, we will present the statistics and details of our final dataset. DL-Bench consists of 520 instances of AI and DL data points (filtered from over 2,000 raw data points). The data is curated from 30 GitHub repositories (selected from an initial pool of 160 related repositories).

As mentioned in Section III, to ensure the accuracy and consistency of the labeling process, four separate annotators were involved. Each instance was analyzed and labeled independently by at least three annotators and a majority voting system was used to minimize bias.

[2]...



(5.1) Pipeline stages (5.2) Task types (5.3) Data types

Fig. 5: Distribution of data points

Implement function **`_tokenize`** that accepts a string **`text`** and returns a list of **token** strings. The method should preprocess the text, encode it using a **SentencePiece model.**

Fig. 6: An example prompt for Text data type

Create a Python function **`draw_bounding_boxes`** that draws bounding boxes on an image tensor. The function should accept an **image** tensor, **bounding boxes**, and optional parameters for **labels** and **colors**, then return the **image with bounding boxes drawn**.

Fig. 7: An example of Image data type

To ensure an accurate evaluation of code generation techniques under test, each prompt instance in DL-Bench is accompanied by at least three test cases (six test cases on average).

One of DL-Bench's contributions is the categories that we assign for each data point. As mentioned in Section III, each data point is assigned a label for which stage of the ML pipeline it belongs to, a label for which ML task it helps solve, and a label for the type of input data. This information enables users of our benchmark to perform an in-depth analysis of their proposed technique with respect to multiple ML-specific aspects. We will demonstrate this in our empirical study presented in Section VI later.

Fig 5 represents the distribution of DL-Bench's data in each categorization. In terms of the stages in the ML pipeline (Fig 5.1), our dataset well covers the five stages of the ML pipeline with the pre/post-processing stage having the most (210) representative samples. Fig 4 lists the prompt to generate a pre/post-processing "draw_point2d" function that can be used to highlighting key points of interest in output images. The model construction stage contains the second most (119) samples such as the one shown in Fig 3. This example shows the prompt to generate the "__init__" method for a fully connected neural network (FCNN). Other ML stages have an equal share of samples. This indicates a balanced dataset that covers all ML stages.

Most of our data serve more than one ML task type, hence 328 (over 63%) instances are labeled as *General* as shown in Fig 5.2.. For example, Fig 8 shows **to_image** function handles data type conversions and pre-processing to standardize image inputs, without performing any specific machine learning task. However, for the cases that serve a specific ML task, our dataset covers all ML tasks evenly with 14 to 60 instances each. Among these, the classification task has the most representative of 60 data points. For example, Fig 9 shows a classification task, calculating precision, recall, and F1 scores for both duplicate and non-duplicate file pairs to evaluate the performance of a classification model. On the other hand, The regression task is not as popular with only 14 data points.

Image data is the most popular input data type with 238 instances (nearly 46%) as shown in Fig 5.3. In some cases where the input data to the function is missing or not the input to the model, we categorize them into the Others category which contains 168 instances. An example of such cases is presented in Fig 3, where the initialization method constructs a new neural network model, however, information on the input

Create a Python function **`to_image`** that accepts an input of type `Union[torch.Tensor, PIL.Image.Image, np.ndarray]` and returns a **`tv_tensors.`**`Image`. The function should check the input type and convert it accordingly: ...

Fig. 8: Example of General Task

Create a Python function `classification_metrics` that takes `ground_truth` and `retrieved` dictionaries and returns per class precision, recall, and F1 scores. Class 1 is assigned to duplicate file pairs while class 0 is for non-duplicate file pairs.

Fig. 9: An example of Classification Task.

type of such networks is not available. Textual data has the least instances since most of the time, textual data is tokenized and presented as either a data array or general tensor.

V. Preliminary Study Design

To demonstrate the potential of DL-Bench and the depth of analysis that can be done on DL-Bench, we perform a preliminary study on how well state-of-the-art LLM models can perform on DL-Bench benchmark. We choose the top LLMs: GPT-4o, Claude3.5 sonet, LLama 3.1 70B, and Mistral 7B. The LLMs are evaluated based on the commonly used passing rate, pass@k, which measures the likelihood that at least one of the top k-generated solutions passes all test cases [42].

A. Experimental Settings

To minimize non-determinism and improve reproducibility, we set the temperature to zero for LLMs[43]. This allowed us to accurately measure the accuracy and quality of the generated code across various DL tasks, offering deeper insights into the LLM's strengths and weaknesses.

For each project, DL-Bench provides a Docker image that includes all required dependencies. These images ensure that test cases are runnable and the evaluations are easily replicable on other systems.

B. Research Questions

To systematically analyze the applicability of DL-Bench, we investigate the following research questions:

- **RQ1—What are the performances of state-of-the-art (SOTA) LLMs on DL/ML code generation tasks?** This RQ aims to analyze how well various SOTA LLMs such as GPT-4o, Claude 3.5 sonet, LLama 3.1, and Mistral 7B perform on ML/DL code generation benchmarks (i.e., prior benchmark DS-1000 and our proposed DL-Bench). For samples in DL-Bench, we also analyzed the typical errors and test failures that generated code triggered. Such analysis provides insight into pitfalls that LLMs often make when performing DL-specific code generation.
- **RQ2—Which stages in the ML pipeline pose a greater challenge for SOTA LLMs?** In this RQ, we assess if code in different stages of the ML pipeline is harder or easier for the SOTA LLMs to generate. This analysis is made possible by our categorization of ML stages (i.e., pre/post-processing, model construction, training, inference, and evaluation & metrics).Such metadata provides an opportunity to perform deeper analyses of the LLMs DL code generation performance.
- **RQ3—Are different ML task types easier or harder to generate code for?** Different ML tasks present different coding challenges and this RQ aims to demonstrate our ML task categorization enables deeper analyses. E.g., by investigating the complexity of each type and examining how the LLMs perform across these diverse categories, we aim to demonstrate DL-Bench's ability in providing valuable insights into areas where LLMs excel and where further improvements are needed.
- **RQ4—Do the various required input data types have any effect on how LLMs generate DL code?** This RQ aims to investigate the performance of LLMs across different input types, including image, text, and tabular data. Specifically, we evaluate how effectively LLMs process and analyze each specified type of input data and, in the process, identify strengths and weaknesses in their ability to handle diverse data formats.

TABLE II: Pass@1 (%) scores for various SOTA LLMs on DS-1000 and DL-Bench. *DS-1000 only publishes results for GPT-4o.

Benchmark	LLM	Pass@1
DS-1000*	GPT-4o	51
DLEval	GPT-4o	31
	Claude 3.5 sonet	28
	LLama 3.1 70B	21
	Mistral 7B	15

VI. Study Results

A. RQ1: What are the performances of SOTA LLMs on DL/ML code generation tasks?

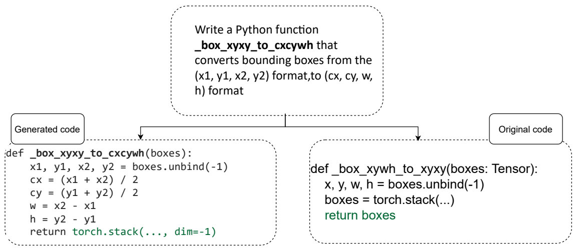
In this RQ, we investigate how the existing ML code generation benchmark (ds-1000)and DL-Bench evaluate SOTA LLMs. Table II shows the pass@1 metric of SOTA LLMs on those benchmarks. To focus on demonstrating DL-Bench's ability to evaluate existing LLMs, we intentionally avoided using specialized prompt strategies, opting instead for vanilla prompts to assess the model's baseline performance. However, the use of advanced prompt engineering strategies could yield different results and this will be interesting future usage of DL-Bench. For DS-1000, the author only reports the result for GPT-4o so we only include that result in our comparison.

Our analysis underscores that even the most advanced model such as GPT-4o struggles with ML/DL-specific code generation. Specifically, GPT-4o achieves only 51% and 31% Pass@1 scores on DS-1000 and DL-Bench respectively. Also, when comparing between DS-1000 and DL-Bench, our benchmark is more challenging as the SOTA LLM get a much lower Pass@1 score, SOTA LLMs in other categories are behind GPT-4o with performance as low as 15% for the tiny Mistral 7B model on DL-Bench. The overall weak performance of these models highlights the ongoing challenges in generating reliable, executable ML/DL code which supports the need for more benchmarks in this domain.

Overall, GPT-4o's pass@k is 31%, however, to further assess GPT's performance, we also computed the average passing rate across functions under test. We found that the average passing rate for GPT-4o is higher at 40%. This suggests that while only 31% of generated functions pass all test cases, there are more functions that could pass some test

TABLE III: Pass@1 scores (%) for different LLMs in different stages on DL-Bench

Model	Pre/Post Processing	Model Construction	Training	Inference	Evaluation & Metrics
GPT-4o	**33**	**26**	**31**	30	**32**
Claude 3.5 sonet	31	22	28	30	29
LLama 3.1 70B	22	14	21	29	24
Mistral 7B	14	11	16	26	19



Write a Python function **_box_xyxy_to_cxcywh** that converts bounding boxes from the (x1, y1, x2, y2) format,to (cx, cy, w, h) format

```
def _box_xyxy_to_cxcywh(boxes):
    x1, y1, x2, y2 = boxes.unbind(-1)
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    return torch.stack(..., dim=-1)
```

```
def _box_xywh_to_xyxy(boxes: Tensor):
    x, y, w, h = boxes.unbind(-1)
    boxes = torch.stack(...)
    return boxes
```

Fig. 10: Pre/post processing example: Converting formats of bounding boxes

cases. This indicates that with additional insights and analysis, such partially valid cases can be improved to be valid.

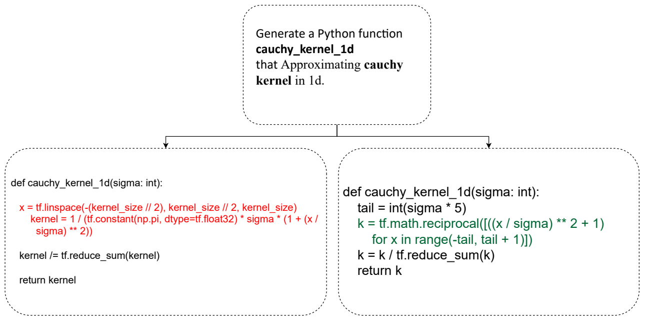
For example, more advanced prompt techniques can be used to improve the result. Such techniques often require additional domain-specific information from deeper analyses that DS-1000 could not provide. In the later RQs, we will demonstrate how DL-Bench can be used to conduct deep analyses so useful insights can be extracted and more sophisticated prompting techniques can be developed to better tackle the task of generating DL-specific code.

Finding 1: Our evaluation indicates that current LLMs struggle to generate correct, executable code for ML/DL tasks. Although GPT-4o is the strongest among the models tested, it still falls short of meeting practical standards (Pass@k score of only 31%). Prior benchmarks such as DS-1000 pose some challenges to the top SOTA LLM but are still not as challenging as DL-Bench. A deeper analysis is needed (which DL-Bench can provide) to extract insight into improving prompting techniques.

B. RQ2: Which stages in the ML pipeline pose a greater challenge for SOTA LLMs?

In this RQ, we conduct a deeper analysis of the challenges of generating DL-specific code for specific stages in the pipeline (enabled by. DL-Bench provided categorization) Table III presents the Pass@1 scores that each LLM achieve for code in each ML/DL pipeline stage. Our result shows that the best SOTA LLM—GPT-4o—outperforms all of the other LLMs in all stages. Claude 3.5 sonet is the closest second, where in *Inference* category, it is level with GPT-4o.

Pre/post Processing stages often require many small but important data transformation tasks thus such codes are the most available. Hence, DL-Bench collected the most data in this category (210/520). Code in this category prepares and cleans input data for the model and formats the model's output. For example, the prompt and reference code in Fig 10

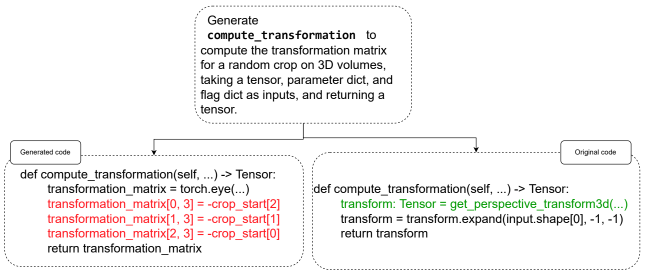


Generate a Python function **cauchy_kernel_1d** that Approximating **cauchy kernel** in 1d.

```
def cauchy_kernel_1d(sigma: int):

  x = tf.linspace(-(kernel_size // 2), kernel_size // 2, kernel_size)
     kernel = 1 / (tf.constant(np.pi, dtype=tf.float32) * sigma * (1 + (x /
       sigma) ** 2))

  kernel /= tf.reduce_sum(kernel)

  return kernel
```

```
def cauchy_kernel_1d(sigma: int):
    tail = int(sigma * 5)
    k = tf.math.reciprocal([((x / sigma) ** 2 + 1)
       for x in range(-tail, tail + 1)])
    k = k / tf.reduce_sum(k)
    return k
```

Fig. 11: Model Construction example



Generate **compute_transformation** to compute the transformation matrix for a random crop on 3D volumes, taking a tensor, parameter dict, and flag dict as inputs, and returning a tensor.

```
def compute_transformation(self, ...) -> Tensor:
    transformation_matrix = torch.eye(...)
    transformation_matrix[0, 3] = -crop_start[2]
    transformation_matrix[1, 3] = -crop_start[1]
    transformation_matrix[2, 3] = -crop_start[0]
    return transformation_matrix
```

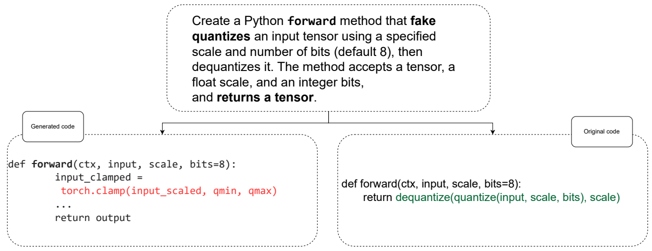
```
def compute_transformation(self, ...) -> Tensor:
    transform: Tensor = get_perspective_transform3d(...)
    transform = transform.expand(input.shape[0], -1, -1)
    return transform
```

Fig. 12: Segmentation Task which GPT-4o failed.

demonstrate a straightforward task of converting the format of the bounding boxes which is common in image-processing ML systems.

Our result shows that LLMs have the most success in generating code for the Pre/post Processing stages. One possible reason for the higher Pass@1 scores is that the models might have more changes to learn from a vast set of samples in this category.

On the other hand, LLMs struggle to generate code for the *Model Construction* stage with the lowest Pass@1 scores. This is because the code for this stage is more complex, very project-specific, and often longer. For example, Fig 11 shows the prompt and the reference code corresponding to a model construction code that GPT-4o failed to generate code for. The main reason is that the code involves utilizing various libraries and constructing a model in a non-standardized way (i.e., with significant variations in structure and organization). Since these tasks appear less frequently in the development pipeline, LLMs are exposed to fewer examples, which diminishes their ability to generate code for these stages.



Create a Python **forward** method that **fake quantizes** an input tensor using a specified scale and number of bits (default 8), then dequantizes it. The method accepts a tensor, a float scale, and an integer bits, and **returns a tensor**.

```
def forward(ctx, input, scale, bits=8):
        input_clamped =
         torch.clamp(input_scaled, qmin, qmax)
        ...
        return output
```

```
def forward(ctx, input, scale, bits=8):
     return dequantize(quantize(input, scale, bits), scale)
```

Fig. 13: An example that GPT is failed to generate the code

Fig 13 presents another case where the LLM struggled, this time with generating a function in the Training stage. Specifically, the forward function for each class can vary significantly, introducing further complexity. These variations make it difficult for the LLM to generalize and learn how to generate the correct code, as it encounters different implementations depending on the context. Perhaps, an interactive prompt technique can help clarify the context or some additional domain-specific information that could be added as a few-step learning to improve the models' performance.

Finding 2: Our study shows that LLMs perform best (e.g., GPT-4o's Pass@1 score is 34%) in Pre/post Processing stages because code in such stages involves common, repetitive tasks, making them easier to learn and generate. In contrast, the Model Construction stage has the lowest scores (e.g., GPT-4o's Pass@1 is 26%) due to its complexity, variability across projects, and the need to integrate multiple libraries. With more details analysis that DL-Bench can enable, future work could develop prompting techniques that focus on providing LLMs with the appropriate context and information to help improve LLMs' performance.

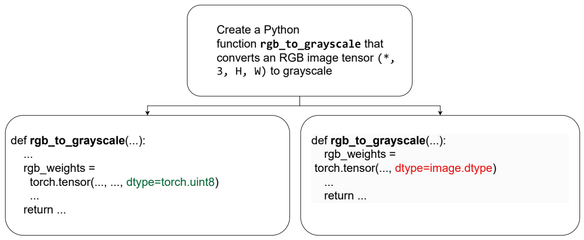
C. RQ3: Are different ML task types easier or harder to generate code for?

In this RQ, we demonstrate how DL-Bench's categorization of ML task types can enable deeper analysis of LLMs code generation and may provide additional insight that can help research proposed more accurate prompting techniques and models. Specifically, we made use of our categorization of ML/DL task types: *Classification*, *Regression*, *Object Detection*, *Image Segmentation*, *Time Series Prediction*, *Recommendation*, and *General*.

There is a significant disparity in the Pass@1 score of generated code across the task types. Notably, the scores for *Recommendation* tasks were the highest among all LLMs, with the best score of 58% with GPT-4o. On the other end of the scale, **Image Segmentation** tasks's scores are the lowest consistently across all LLMs. These results indicate that each task type has its own characteristics that LLMs can or not yet capture.

Specifically, we suspect that image segmentation code is harder to generate due to its close relation to image processing. Specifically, segmentation tasks involve dividing an image into distinct regions, which can vary greatly based on image type, number of channels, and image resolution which present significant challenges to even the top-of-the-line LLMs such as GPT-4o. For example, Fig 12 shows an example where the generated code incorrectly assumes that cropping can be represented by a simple translation matrix.

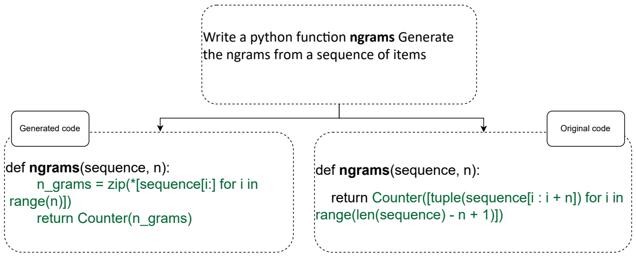
Recommendation codes, on the other hand, generally involve structured data, such as user-item interactions, which are more predictable compared to image data. The limited variation in data structures and patterns across different recommendation tasks makes it easier for GPT-4o to generate accurate code. Specifically, Recommendation systems typically


Create a Python function **rgb_to_grayscale** that converts an RGB image tensor (*, 3, H, W) to grayscale

```
def rgb_to_grayscale(...):
  ...
  rgb_weights =
   torch.tensor(..., ..., dtype=torch.uint8)
   ...
  return ...
```

```
def rgb_to_grayscale(...):
  rgb_weights =
 torch.tensor(..., dtype=image.dtype)
  ...
  return ...
```

Fig. 14: An Example of difference variance of Image as data.


Write a python function **ngrams** Generate the ngrams from a sequence of items

```
def ngrams(sequence, n):
     n_grams = zip(*[sequence[i:] for i in
range(n)])
     return Counter(n_grams)
```

```
def ngrams(sequence, n):

   return Counter([tuple(sequence[i : i + n]) for i in
range(len(sequence) - n + 1)])
```

Fig. 15: Text data example: Defining n-gram extraction of a list of tokenized text

rely on common algorithms such as collaborative filtering or matrix factorization, both of which follow relatively standardized patterns. This standardization likely enables GPT-4o to "memorize" these patterns (one example is shown in Fig 16).

Finding 3: Different ML/DL tasks have varying levels of complexity which significantly influence the code generation ability of LLMs. LLMs achieved the highest scores (up to 58% with GPT-4o) for recommendation code due to their predictable input structure and patterns. Conversely, they struggled with code for image segmentation tasks (up to only 19% Pass@1 score) involving complex image data requiring pixel-level understanding and generalization across variable inputs. DL-Bench's categorization provides additional insights to improve DL code generation techniques.

D. RQ4: Do the various required input data types have any effect on how LLMs generate DL code?

This RQ aims to investigate if different input data type have different effect on how well LLMs generate code. DL-Bench enables this analysis by providing a categorization of input data types: *mage*, *Text*, **Structured Array**, and *Others*. By comparing their performance across these input types, the study evaluates the versatility and limitations of LLMs in dealing with varied data sources. Table V shows the Pass@1 of all LLMs across the different types of input data. Specifically, performance for image-related tasks is the lowest (only up to 24% for GPT-4o). This can be attributed to the inherent complexity and lack of consistent structure in image data, such as varying shapes, resolutions, and channel configurations (e.g., grayscale vs. RGB). Fig 14 shows an example where GPT-4o failed to generate the correct type torch.unit8.

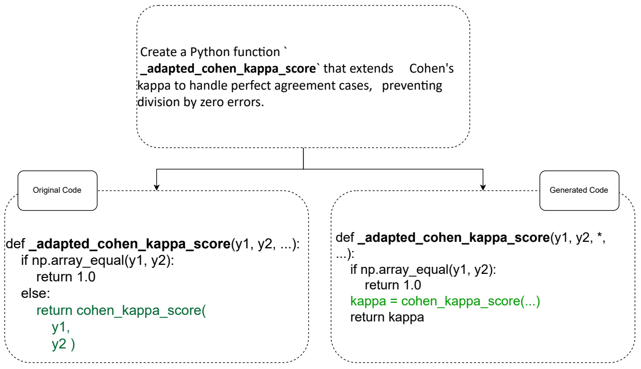
On the other hand, our results show that textual input data code generation exhibits better performance (up to 32%). We assume that most textual input data are tokenized and

TABLE IV: Pass@1 (%) scores for different ML/DL tasks on DL-Bench

Model	Classification	Regression	Object Detection	Image Segmentation	Time Series Prediction	Recommendation	General
GPT-4o	**30**	**36**	**28**	**19**	**33**	**58**	**31**
Claude 3.5 sonet	**32**	35	30	21	29	47	27
LLama 3.1 70B	28	22	17	14	31	40	19
Mistral 7B	24	21	11	11	13	29	13

TABLE V: Pass@1 (%) scores across various input data types on DL-Bench

Model	Image	Text	Structured Array	Others
GPT-4o	**25**	**32**	**29**	**40**
Claude 3.5 sonet	**25**	30	24	33
LLama 3.1 70B	18	30	19	26
Mistral 7B	8	23	13	25


Create a Python function `_adapted_cohen_kappa_score` that extends Cohen's kappa to handle perfect agreement cases, preventing division by zero errors.

```
def _adapted_cohen_kappa_score(y1, y2, ...):
    if np.array_equal(y1, y2):
        return 1.0
    else:
        return cohen_kappa_score(
            y1,
            y2 )
```

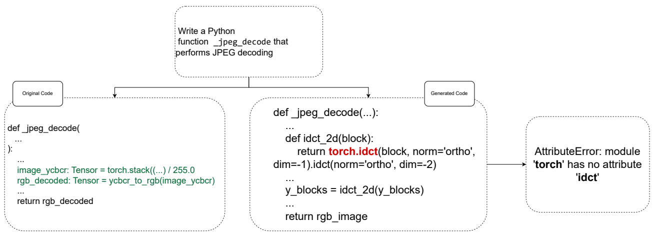
```
def _adapted_cohen_kappa_score(y1, y2, *,
...):
    if np.array_equal(y1, y2):
        return 1.0
    kappa = cohen_kappa_score(...)
    return kappa
```

Fig. 16: Structured Array example: Generating function to compute adapted Cohen Kappa score based on an array

converted before being processed in the actual DL model, this makes functions that deal directly with textual input data quite standard and easier to generate. Fig 15 shows an example of a text-related prompt where GPT-4o can generate the correct code. Specifically, the prompt asks LLMs to convert to n-grams from a sequence of tokenized text which belong to the pre-processing stage.

The structured array category shows slightly higher performance (up to 29% with GPT-4o) compared to image data. This is because, structured data inherently provides a clearer and more organized format, reducing the level of reasoning required by the model. As a result, the model can more easily generate accurate code for table-related tasks, as opposed to the more unstructured and abstract nature of image-based tasks. This suggests that structured data simplifies the generation process,

Fig 16 provides an example of structured array data where GPT successfully generates accurate code. The function _adapted_cohen_kappa_score, designed to extend Cohen's kappa score by handling the special case of perfect agreement and preventing a division by zero error, operates on two Numpy arrays y1 and y2. GPT can generate code for this example efficiently because the input data consists


Write a Python function `_jpeg_decode` that performs JPEG decoding

```
def _jpeg_decode(
  ...
):
    ...
    image_ycbcr: Tensor = torch.stack((...) / 255.0
    rgb_decoded: Tensor = ycbcr_to_rgb(image_ycbcr)
    ...
    return rgb_decoded
```

```
def _jpeg_decode(...):
    ...
    def idct_2d(block):
        return torch.idct(block, norm='ortho',
dim=-1).idct(norm='ortho', dim=-2)
    ...
    y_blocks = idct_2d(y_blocks)
    ...
    return rgb_image
```

```
AttributeError: module
'torch' has no attribute
'idct'
```

Fig. 17: Wrong usage of third-party library.

of two Numpy arrays, which are are commonly used in numerical computations and have a well-defined structure that is predictable for the model. Both y1 and y2 share the same Numpy structure, making it easier to generate logic based on well-known Numpy operations.

Finding 4: This RQ demonstrate the kind of detailed analysis that future research can perform with DL-Bench. Our preliminary study shows that image data had the lowest performance (Pass@1 score of only up to 24%) due to the complexity and inconsistency of its input structure. In contrast, textual data tasks yielded higher performance (up to 32%), likely due to more deterministic coding that often belongs to the pre-processing stage.

VII. Discussion

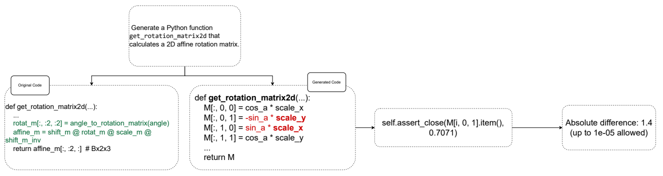
In this section, we will discuss some qualitative analysis and discuss three common types of errors that LLM-generated code often trigger during test execution:

A. Incorrect Usage of Third-Party Libraries

Third-party libraries (e.g., PyTorch, TensorFlow, and NumPy) are essential to writing ML/DL code as oftentimes it is not effective and efficient to write everything from scratch, especially with so many available popular ML/DL APIs. However, as illustrated in Fig 17, LLMs sometimes fail to use these libraries correctly. In this case, the model attempts to call *torch.idct*, which is not implemented in PyTorch. One possible fix is to provide more context concerning third-party libraries. For example, one could provide a hint to LLMs to use *scipy* instead which would result in the use of *scipy.fftpack.idct(x.numpy(), norm=norm)* instead.

B. Incorrect Operations on Input Parameters

Not only do the API calls need to be correct, but the input parameters also need to be appropriately processed. For example, LLMs sometimes perform incorrect operations on input parameters. For example, in Fig 18, GPT-4o applies **scale_x** only to the cosine whereas the scaling factors **scale_x**


Generate a Python function get_rotation_matrix2d that calculates a 2D affine rotation matrix.
Original Code
Generated Code

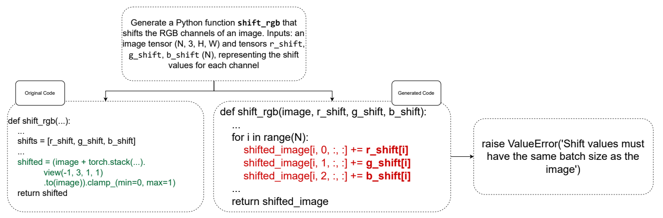
```
def get_rotation_matrix2d(...):
    ...
    rotat_m[:, :2, :2] = angle_to_rotation_matrix(angle)
    affine_m = shift_m @ rotat_m @ scale_m @
shift_m_inv
    return affine_m[:, :2, :]  # Bx2x3
```

```
def get_rotation_matrix2d(...):
    M[:, 0, 0] = cos_a * scale_x
    M[:, 0, 1] = -sin_a * scale_y
    M[:, 1, 0] = sin_a * scale_x
    M[:, 1, 1] = cos_a * scale_y
    ...
    return M
```

```
self.assert_close(M[i, 0, 1].item(),
0.7071)
```

```
Absolute difference: 1.4
(up to 1e-05 allowed)
```

Fig. 18: Incorrect processing of parameters: The axes scales need to be applied to both sin and cos


Generate a Python function shift_rgb that shifts the RGB channels of an image. Inputs: an image tensor (N, 3, H, W) and tensors r_shift, g_shift, b_shift (N), representing the shift values for each channel
Original Code
Generated Code

```
def shift_rgb(...):
    ...
    shifts = [r_shift, g_shift, b_shift]
    ...
    shifted = (image + torch.stack(...).
            view(-1, 3, 1, 1)
            .to(image)).clamp_(min=0, max=1)
    return shifted
```

```
def shift_rgb(image, r_shift, g_shift, b_shift):
    ...
    for i in range(N):
        shifted_image[i, 0, :, :] += r_shift[i]
        shifted_image[i, 1, :, :] += g_shift[i]
        shifted_image[i, 2, :, :] += b_shift[i]
    ...
    return shifted_image
```

```
raise ValueError('Shift values must have the same batch size as the image')
```

Fig. 19: Mismatching data shapes: shifting variables need to be broadcasted to the image shape

and **scale_y** should be applied uniformly to both the sine and cosine components of the rotation matrix. This results in improper scaling along the axes and triggers a test failure.

C. Incorrect Assumptions About Input and Output Shapes

Input/output shapes are also hard for LLMs to handle. Specifically, as shown in Fig 19, when shifting values *(r_shift, g_shift, b_shift*) are one-dimensional tensors with shape (N) they need to be broadcastable to the image channels with shape (N, H, W) in order for the += operation to work. The GPT-4o incorrectly assumes that each shift value can be applied directly to all pixels in the image channel, causing a shape mismatch. Additional few-steps learning specifically focused on shape correction would help in cases where the shift values are reshaped to (N, 1, 1) using *.view(N, 1, 1).*

VIII. Related Works

A. Code Generation Benchmarks

The HumanEval benchmark [24] is foundational for assessing Python code functional correctness from docstrings, with 164 curated problems. AiXBench [44] expands to Java, adding 175 automated and 161 manually reviewed samples with a new correctness metric. MultiPL-E [45] broadens HumanEval's reach, supporting 18 languages for multi-language code evaluation. For program synthesis, LLM performance is measured on MBPP and MathQA-Python [11], covering 974 tasks for beginners and 23,914 complex problems, showing improved model capabilities with size and fine-tuning, and potential for dialog-based solution refinement.

The Spider benchmark [14] focuses on generating SQL queries from natural language with 10,000 questions and 5,000 queries across 200 databases, emphasizing generalization to unseen databases. CoderEval [25] evaluates real-world code generation in Python and Java, and APPS [10] presents 10,000 problems across to test models on complex algorithmic coding.

B. Code Generation for AI

Shin et al.[46] examined neural code generation models for ML tasks, noting the distinct challenges compared to general programming and the need for specialized benchmarks. Unlike Shin's work, our benchmark targets AI and deep learning tasks with structured test cases and categorized instances for a focused evaluation of code generation in these fields.

Similarly, DS-1000 [17] is a data science code generation benchmark with a thousand real-world problems across seven Python libraries. Our benchmark differs by focusing on deep learning, categorizing data by pipeline steps, data types, and task types, and providing function-level granularity from GitHub sources instead of StackOverflow.

IX. Threats and Validity

Our experiments utilized non-deterministic models, even with the temperature parameter set to zero. While a lower temperature reduces randomness, it does not fully eliminate variability in the models' outputs [47], [48]. Moreover We sourced data from various repositories related to deep learning and AI but did not include all possible repositories or tags. Expanding the dataset could capture a wider range of use cases and code patterns.

Data labeling was performed by three independent reviewers, achieving a strong inter-rater reliability. Despite this, some labeling conflicts persisted and were addressed through discussions to reach a consensus. However, not all discrepancies could be fully resolved. Also, We used the pass@k metric to evaluate model performance, but passing provided test cases does not guarantee that the generated code handles all potential edge cases or is semantically correct in every scenario [49].

X. Conclusion

In this paper, we introduce DL-Bench, a benchmark for deep learning tasks related to code generation. The dataset comprises 520 instances, gathered from the most starred and recently updated GitHub repositories. We categorize the data based on the pipeline stage, data type, and task type. Multiple reviewers ensure the accuracy of our labels. Additionally, we conduct experiments using GPT-4o, Claude 3.5 sonet, Llama 3.1 70b, and Mistral 7b, categorizing pass@1 accuracy across different criteria. We also provide examples demonstrating the distribution and diversity of our benchmark.

References

[1] N. F. Hordri, S. S. Yuhaniz, and S. M. Shamsuddin, "Deep learning and its applications: A review," in *Conference on Postgraduate Annual Research on Informatics Seminar*, 2016, pp. 1–5.

[2] A. Kamilaris and F. X. Prenafeta-Boldú, "Deep learning in agriculture: A survey," *Computers and electronics in agriculture*, vol. 147, pp. 70–90, 2018.

[3] J. C. B. Gamboa, "Deep learning for time-series analysis," *arXiv preprint arXiv:1701.01887*, 2017.

[4] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software engineering challenges of deep learning," in *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2018, pp. 50–59.

[5] S. Park, A. Y. Wang, B. Kawas, Q. V. Liao, D. Piorkowski, and M. Danilevsky, "Facilitating knowledge sharing from domain experts to data scientists for building nlp models," in *Proceedings of the 26th International Conference on Intelligent User Interfaces*, 2021, pp. 585–596.

[6] S. Singaravel, J. Suykens, H. Janssen, and P. Geyer, "Explainable deep convolutional learning for intuitive model development by non–machine learning domain experts," *Design Science*, vol. 6, p. e23, 2020.

[7] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[9] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[10] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[11] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *ArXiv*, vol. abs/2108.07732, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:237142385

[12] R. Agashe, S. Iyer, and L. Zettlemoyer, "JuICe: A large scale distantly supervised dataset for open domain context-based code generation," 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)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5436–5446. [Online]. Available: https://aclanthology.org/D19-1546

[13] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[14] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman *et al.*, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *arXiv preprint arXiv:1809.08887*, 2018.

[15] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[16] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[17] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[18] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 333–336. [Online]. Available: https://doi.org/10.1145/2668930.2688819

[19] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong *et al.*, "Understanding llms: A comprehensive overview from training to inference," *arXiv preprint arXiv:2401.02038*, 2024.

[20] H. El-Amir and M. Hamdy, *Deep learning pipeline: building a deep learning model with TensorFlow*. Apress, 2019.

[21] D. J. E. Waibel, S. Shetab Boushehri, and C. Marr, "Instantdl: an easy-to-use deep learning pipeline for image segmentation and classification," *BMC bioinformatics*, vol. 22, pp. 1–15, 2021.

[22] Y. Zhou, H. Chen, Y. Li, Q. Liu, X. Xu, S. Wang, P.-T. Yap, and D. Shen, "Multi-task learning for segmentation and classification of tumors in 3d automated breast ultrasound images," *Medical Image Analysis*, vol. 70, p. 101918, 2021.

[23] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 689–696.

[24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Pondé, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235755472

[25] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[26] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[27] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[28] D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak, "Repofusion: Training code models to understand your repository," *arXiv preprint arXiv:2306.10998*, 2023.

[29] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[30] M. AlMarzouq, A. AlZaidan, and J. AlDallal, "Mining github for research and education: challenges and opportunities," *International Journal of Web Information Systems*, vol. 16, no. 4, pp. 451–473, 2020.

[31] N. Tran, H. Tran, S. Nguyen, H. Nguyen, and T. Nguyen, "Does bleu score work for code migration?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 165–176.

[32] M. Madeja, J. Porubän, M. Bačíková, M. Sulír, J. Juhár, S. Chodarev, and F. Gurbál', "Automating test case identification in java open source projects on github," *arXiv preprint arXiv:2102.11678*, 2021.

[33] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[34] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[35] Z. Chen and S. Moscholios, "Using prompts to guide large language models in imitating a real person's language style," *arXiv preprint arXiv:2410.03848*, 2024.

[36] R. Xie, "Frontiers of deep learning: From novel application to real-world deployment," *arXiv preprint arXiv:2407.14386*, 2024.

[37] M. Assi, S. Hassan, and Y. Zou, "Unraveling code clone dynamics in deep learning frameworks," *arXiv preprint arXiv:2404.17046*, 2024.

[38] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN computer science*, vol. 2, no. 3, p. 160, 2021.

[39] P. K. Vinodkumar, D. Karabulut, E. Avots, C. Ozcinar, and G. Anbarjafari, "A survey on deep learning based segmentation,

detection and classification for 3d point clouds," *Entropy*, vol. 25, no. 4, 2023. [Online]. Available: https://www.mdpi.com/1099-4300/25/4/635

[40] N. Manakitsa, G. S. Maraslidis, L. Moysis, and G. F. Fragulis, "A review of machine learning and deep learning for object detection, semantic segmentation, and human action recognition in machine and robotic vision," *Technologies*, vol. 12, no. 2, 2024. [Online]. Available: https://www.mdpi.com/2227-7080/12/2/15

[41] A. Zapf, S. Castell, L. Morawietz, and A. Karch, "Measuring inter-rater reliability for nominal data–which coefficients and confidence intervals are appropriate?" *BMC medical research methodology*, vol. 16, pp. 1–10, 2016.

[42] Z.-C. Lyu, X.-Y. Li, Z. Xie, and M. Li, "Top pass: Improve code generation by pass@ k-maximized code ranking," *arXiv preprint arXiv:2408.05715*, 2024.

[43] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[44] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei, "Aixbench: A code generation benchmark dataset," *arXiv preprint arXiv:2206.13179*, 2022.

[45] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.

[46] J. Shin, M. Wei, J. Wang, L. Shi, and S. Wang, "The good, the bad, and the missing: Neural code generation for machine learning tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–24, 2023.

[47] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

[48] Y. Song, G. Wang, S. Li, and B. Y. Lin, "The good, the bad, and the greedy: Evaluation of llms should not ignore non-determinism," *arXiv preprint arXiv:2407.10457*, 2024.

[49] N. Shiri Harzevili, M. M. Mohajer, M. Wei, H. V. Pham, and S. Wang, "History-driven fuzzing for deep learning libraries," *ACM Transactions on Software Engineering and Methodology*.