Experiment Report

I. Project Background

In today's rapidly advancing technological era, the research and development of programming language models have become a frontier topic in the field of artificial intelligence. With the exponential growth in software development demands, the application prospects of automated code generation technology are becoming increasingly broad. The development of the `deepseek-ai/deepseek-coder-1.3b-instruct` model aims to advance this field by leveraging the powerful capabilities of large-scale pre-trained language models to significantly enhance the efficiency and accuracy of code generation.

The core objective of this project is to systematically evaluate and optimize the performance of the `deepseek-ai/deepseek-coder-1.3b-instruct` model in code generation tasks. Through a carefully designed experimental process, we aim to deeply analyze the model's initial performance across different programming languages and explore the potential for improving its performance on specific tasks through fine-tuning techniques. The success of the project will provide software developers with more powerful tools, significantly reducing manual coding time and error rates, thereby improving development efficiency and software quality.

II. Experiment Overview

This experiment aims to comprehensively evaluate and optimize the `deepseek-ai/deepseek-coder-1.3b-instruct` model, which focuses on code generation tasks and is a large-scale pre-trained language model. The experiment is divided into three main parts: model benchmarking, large model fine-tuning, and model evaluation using CfrystalBLEU. Through these steps, we hope to gain a deep understanding of the model's initial performance and explore the potential for improving its performance on specific tasks through fine-tuning on specific datasets.

III. Model Benchmarking

Environment Configuration

- **Operating System**: Linux Ubuntu, Windows 11
- **Hardware Resources**: NVIDIA GeForce RTX 4090 GPU
- **Programming Language Version**: Python

Testing Method

In this experiment, we used `human_eval` as a benchmarking tool to evaluate the performance of the `deepseek-ai/deepseek-coder-1.3b-instruct` model in Java and Python code generation tasks. `human_eval` is a widely used evaluation framework specifically designed to measure the capabilities of code generation models. It assesses whether the generated code can correctly execute the specified tasks by running it.

Human Eval

`human_eval` is a standard tool for evaluating code generation models. Its core idea is to test the model's code generation capabilities through a set of predefined programming tasks. Each task includes a problem description and one or more test cases. The model needs to generate code based on the problem description and pass the validation of the test cases. The test cases in `human_eval` cover a wide range of application scenarios, including basic algorithm implementations, data structure operations, and complex programming problems, ensuring the comprehensiveness and reliability of the evaluation results.

pass@1

In `human_eval`, `pass@1` is an important evaluation metric. It represents the proportion of tasks where the model successfully passes all test cases with the first generated code snippet. Specifically, `pass@1` calculates the percentage of tasks where the first code snippet generated by the model correctly solves the problem and passes all test cases. This metric reflects the model's ability to generate correct code on the first attempt and is a key standard for measuring the performance of code generation models.

Running Benchmark Model Tests

To run the benchmark model tests, execute the `HumanEval/run_eval.sh` script. Below is the content of the script:

```
export LANG=python  #java
export OUTPUT_DIR=output
export MODEL_PATH="../finetune/models/deepseek-coder-1.3b-instruct"
export CUDA_VISIBLE_DEVICES=0

if [ ! -d "$OUTPUT_DIR" ]; then
  mkdir -p "$OUTPUT_DIR"
fi

COMMAND=("python" "eval_instruct.py"
         "--model" "$MODEL_PATH"
         "--output_path" "${OUTPUT_DIR}/${LANG}.deepseek-coder-1.3b-
instruct.jsonl"
         "--language" "$LANG"
         "--temp_dir" "$OUTPUT_DIR")

echo "Executing command: ${COMMAND[*]}"

"${COMMAND[@]}"
if [ $? -eq 0 ]; then
    echo "Command executed successfully"
else
    echo "Error executing command"
fi
```

Results

Programming Language	pass@1
Python	67.68%
Java	52.53%

The results show that the model performs well in Python code generation, while the accuracy in Java code generation is relatively lower. This may reflect differences in the model's adaptability between different programming languages or be due to a higher proportion of Python code snippets in the training dataset. The `pass@1` results provide a direct quantification of the model's ability to generate correct code on the first attempt.

IV. Large Model Fine-Tuning

When fine-tuning the `deepseek-ai/deepseek-coder-1.3b-instruct` model, we employed the LoRA technique. Below are the key settings and parameters during the fine-tuning process:

Fine-Tuning Target Modules

- **v_proj and q_proj**: These are part of the self-attention mechanism in the Transformer architecture. q_proj (query projection) and v_proj (value projection) are used to linearly transform input vectors into query and value vectors. LoRA adjusts these weight matrices by adding low-rank matrices instead of directly modifying the original parameters in the pre-trained model.

LoRA Parameter Settings

- **LoRA alpha value**: 16
    - **Explanation**: The alpha value acts as a scaling factor in LoRA, determining the extent to which the low-rank matrix affects the original weight matrix. A larger alpha value means the low-rank matrix will more strongly alter the behavior of the original weight matrix.
    - **Benefit**: Setting an appropriate alpha value can help balance the effect of fine-tuning with model stability. In this case, alpha = 16 indicates a desire to introduce as much new knowledge as possible without disrupting the model's original performance.
- **r value (rank)**: 8
    - **Explanation**: The r value determines the size of the low-rank matrix used to adjust the weight matrix. A lower r value means fewer new parameters need to be learned.
    - **Benefit**: Using r = 8 can significantly reduce the number of parameters that need to be optimized, lower the risk of overfitting, and make the model easier to converge.

Fine-Tuning Hyperparameters

Parameter/Configuration	Details
Target Modules	v_proj, q_proj
LoRA alpha value	16
r value (rank)	8
Number of Training Epochs	3
Model Maximum Length	512
Per Device Training Batch Size	4
Per Device Evaluation Batch Size	4
Learning Rate	3e-5
Warmup Steps	100
Logging Steps	1
Learning Rate Scheduler Type	cosine
Gradient Clipping	Enabled
Mixed Precision (fp16)	Enabled
Optimizer	adamw_torch

Dataset

The dataset used for fine-tuning is Hugging Face's `EvolInstruct-Code-80k.json`, which contains code snippets in multiple programming languages, making it well-suited for training code generation models. The dataset has been carefully cleaned and preprocessed to ensure high-quality training samples. In this experiment, we selected 10,000 data entries for fine-tuning and 2,000 entries as the test set to ensure the model receives sufficient diversity in training while keeping computational costs manageable.

Prompt Engineering

To further enhance the code generation capabilities of the `deepseek-ai/deepseek-coder-1.3b-instruct` model, prompt engineering was introduced during the fine-tuning process. Prompts serve as an effective guiding tool, providing the model with additional task context and guidance information to help it better understand task requirements and generate code that meets expectations.

- **Task Description**: Each training sample is accompanied by a brief task description as part of the prompt, clearly explaining the task objectives in natural language.

Fine-Tuning Execution

To run fine-tuning on the Windows operating system, execute the `fineture/run.py` file, and the model will be saved in the `fineture/output` directory. Below is the content of the execution file:

```
import subprocess
import os
```

```
DATA_PATH = "data/EvolInstruct-Code-80k.json"
OUTPUT_PATH = "output"
MODEL_PATH = "models/deepseek-coder-1.3b-instruct"

command = [
    "python", "finetune_deepseekcoder.py",
    "--model_name_or_path", MODEL_PATH,
    "--data_path", DATA_PATH,
    "--output_dir", OUTPUT_PATH,
    "--num_train_epochs", "3",
    "--model_max_length", "512",
    "--per_device_train_batch_size", "4",
    "--per_device_eval_batch_size", "4",
    "--gradient_accumulation_steps", "1",
    "--evaluation_strategy", "no",
    "--save_strategy", "steps",
    "--save_steps", "500",
    "--save_total_limit", "1",
    "--learning_rate", "3e-5",
    "--warmup_steps", "100",
    "--logging_steps", "1",
    "--lr_scheduler_type", "cosine",
    "--gradient_checkpointing",
    "--report_to", "tensorboard",
    "--max_grad_norm", "1.0",
    "--fp16"
]

subprocess.run(command, check=True)
```

Fine-Tuning Time and Loss

- **Training Run Time**: 85,639.1934 seconds
- **Training Samples per Second**: 0.35
- **Training Steps per Second**: 0.088
- **Training Loss**: 0.2120
- **Training Epochs**: 3.0

V. CrystalBLEU Model Evaluation

Technical Background

In code generation tasks, evaluating the quality of generated code is a key challenge. Traditional evaluation methods like BLEU scores, while performing well in natural language processing, have limitations in code generation tasks. The syntactic verbosity and coding conventions of programming languages lead to many n-grams being shared in completely unrelated code snippets, making BLEU scores less effective at distinguishing semantically similar and dissimilar code. To address this issue, CrystalBLEU was proposed as an improved evaluation metric. CrystalBLEU removes trivially shared n-grams that frequently appear in the same language but do not imply semantic similarity before calculating n-gram overlap between code snippets, thereby more accurately evaluating code similarity.

Evaluation Method

After fine-tuning, we evaluated the `deepseek-ai/deepseek-coder-1.3b-instruct` model using the CrystalBLEU metric. CrystalBLEU analyzes a set of code samples to identify frequently occurring n-grams and excludes these trivially shared n-grams during evaluation to increase sensitivity to semantic similarity. We selected 2,000 data entries from Hugging Face's `EvolInstruct-Code-80k.json` dataset for testing and checked the syntactic correctness and logical completeness of the generated code to ensure the reliability of the evaluation results.

Tokenization Process

During the evaluation, we used the tokenizer of the `deepseek-ai/deepseek-coder-1.3b-instruct` model for tokenization. This step ensures compatibility with the input data and improves the accuracy of the evaluation.

CrystalBLEU Evaluation Method

CrystalBLEU is a metric for evaluating the similarity between generated code and reference code. Unlike traditional BLEU scores, CrystalBLEU focuses not only on literal similarity but also on semantic similarity of the code. Its calculation process includes the following steps:

1. **n-gram Extraction**: Extract n-grams from both generated and reference code.
2. **Trivial n-gram Filtering**: Remove trivially shared n-grams that frequently appear in the target programming language.
3. **Similarity Calculation**: Calculate the overlap of filtered n-grams to obtain the CrystalBLEU score.

To improve the accuracy of the evaluation, CrystalBLEU introduces a technique for dynamically selecting trivially shared n-grams. The key to this process is identifying and filtering out n-grams that do not affect semantic similarity in the target programming language.

Dynamic Selection of trivially_shared_ngrams

In the implementation of CrystalBLEU, the dynamic selection of trivially shared n-grams involves the following steps:

1. **n-gram Extraction**: Extract all possible n-grams from both generated and reference code.
2. **Frequency Analysis**: Calculate the frequency of each n-gram across the entire dataset. High-frequency n-grams are usually common structures or keywords of the language.
3. **Dynamic Filtering**: Use the `min_length` parameter to determine which n-grams should be considered trivially shared. Specifically, only consider n-grams with a length greater than or equal to `min_length` to avoid mistakenly treating overly short n-grams (such as single characters or common short words) as meaningful semantic units.
4. **Construct trivially_shared_ngrams**: Based on the frequency analysis results, construct a dictionary `trivially_shared_ngrams` containing all n-grams considered trivially shared and their occurrence counts. This dictionary is used to filter out these n-grams when calculating the CrystalBLEU score.

```
min_length = 2
trivially_shared_ngrams = {
    str(ngram): count for ngram, count in frequencies.most_common(k)
    if len(ngram) >= min_length
}
```

Through this dynamic selection method, CrystalBLEU can more accurately identify and filter out n-grams that do not affect semantic similarity, thereby improving the ability to evaluate the quality of code generation. This method ensures that the evaluation results better reflect the actual semantic similarity of the generated code, rather than just literal similarity.

Results

- **CrystalBLEU Score**: 0.1783

VI. Conclusion

This experiment demonstrated that the `deepseek-ai/deepseek-coder-1.3b-instruct` model performs well in Python code generation without adjustments, while there are shortcomings in Java code generation. After fine-tuning on a multi-language code generation dataset, although the CrystalBLEU score indicates room for improvement, the overall results show that the model has potential in code generation tasks. Future work can focus on the following areas:

1. **Optimize Fine-Tuning Strategies**: Improve the model's generation capabilities by adjusting fine-tuning data and methods.
2. **Develop More Accurate Evaluation Metrics**: Combine other evaluation methods, such as CodeBLEU and semantic similarity analysis, to more comprehensively assess the quality of code generation tasks.
3. **Enhance Syntax and Logic Checks**: Incorporate stricter syntax and logic validation during evaluation to ensure the practicality and correctness of the generated code.

Appendix

Large File Description

Due to the large size of the `model.safetensors` and `optimizer.pt` files, they cannot be uploaded to GitHub. These two files represent the model's weights and the optimizer's state, respectively.