Bridging the Quantum Coding Gap: Instruction-Tuned LLMs for Qiskit

Sixu Chen
Kent State University
schen53@kent.edu

Yuqi Zhang
Kent State University
yzhan135@kent.edu

Qiang Guan
Kent State University
qguan@kent.edu

Abstract

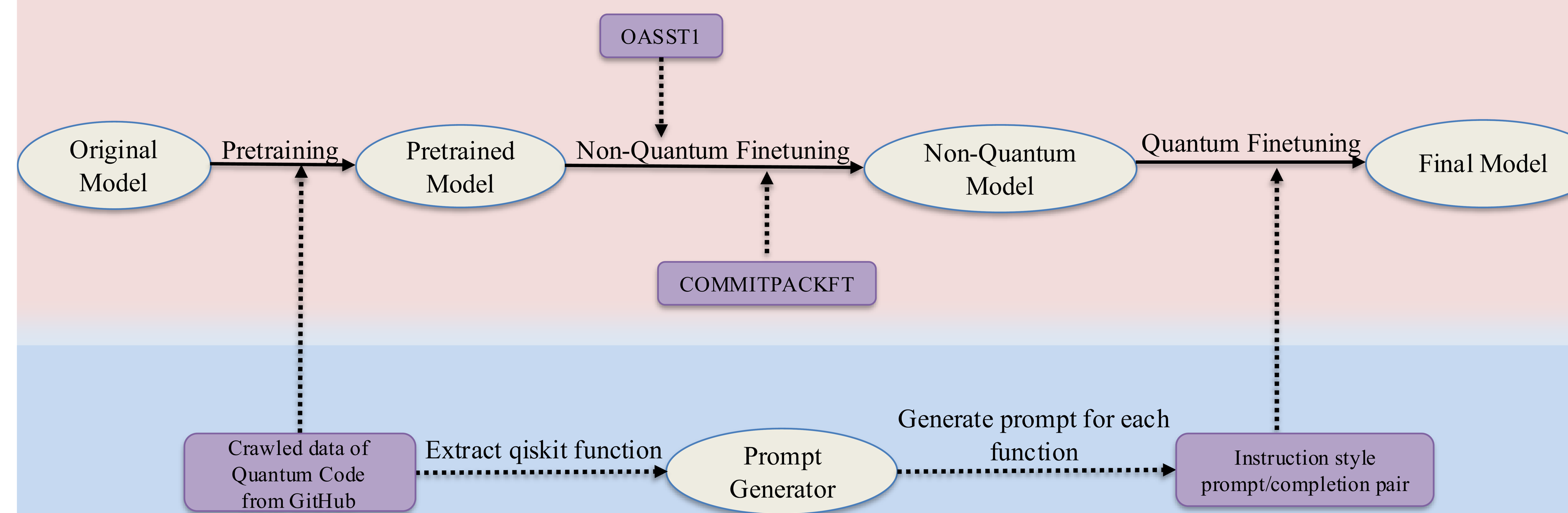
Quantum computing is an emerging and promising field that aims to efficiently solve problems that are intractable for classical computers. However, learning quantum computing remains challenging due to its reliance on mathematical and physical, which creates a high learning barrier and may hinder broader engagement and slow the development and adoption of quantum computing. Large Language Models (LLMs) [1] have shown strong capabilities in code generation, but they perform poorly on quantum programming tasks. To address this, we focus on Qiskit and fine-tune LLMs using quantum code from GitHub, OASST1 [2], and COMMITPACKFT. We further construct instruction-style prompt/completion pairs from real Qiskit functions to improve fine-tuning quality. Evaluations on HumanEval and Quantum HumanEval (QHE) [3] show that our method significantly enhances quantum code generation performance.

Data

- OASST1 dataset is a human-generated, human-annotated assistant-style conversation trees consisting 66,497 conversations in 35 different languages between a user and an assistant. We only use the top response at each level of the conversation tree and filter out the out moralizing assistant answers, which remains 8,587 samples.
- COMMITPACKFT contains 2 gigabytes high quality permissively licensed code instructions across 277 languages. Because Qiskit is implemented in Python, we then randomly sample 5,000 Python instances from COMMITPACKFT, enabling the model to learn Python-specific constraints and idioms relevant to Qiskit.
- We crawled 18.42 MB code which qiskit in the name from GitHub. Then instruction style prompt/completion qiskit code dataset is constructed by extracting the single function from our crawled qiskit code. Then use Llama-3-8B-Instruct to generate purpose description for each functions. Finally, we wrap these descriptions into a prompt in HumanEval's prompt style.
- For evaluation, we adopt we two execution-based datasets: Qiskit HumanEval(QHE) and HumanEval(HE). HumanEval contains 164 Python programming problems and Qiskit HumanEval consisting 151 challenging qiskit code.

.

Methodology



- We first pretrain a language model using quantum-related code crawled from GitHub. This helps the model learn foundational patterns in both natural language and quantum programming.
- To enhance the model's instruction-following ability and strengthen its general programming logic, we fine-tune it on two diverse non-quantum datasets: OASST1 and COMMITPACKFT.
- Finally, we construct instruction-style prompt/completion pairs from real-world Qiskit functions. These are used to fine-tune the model in a task-specific manner, enabling better alignment between natural language instructions and quantum code generation.

Generated Sample

Model	HE	QHE
Qwen2.5-Math-7B	0.58	0.15
qiskit-Qwen2.5-Math-7B	0.61	0.23
AutoCoder_QW_7B	0.89	0.11
qiskit-AutoCoder_QW_7B	0.71	0.18

Table: Evaluation on Qiskit HumanEval (QHE) and HumanEval (HE) via pass@1 score.

- We observe that all models perform substantially better on the HumanEval benchmark compared to the QHE highlighting a clear gap between general purpose code generation and quantum-specific code generation.
- Among all the models, qiskit-Qwen2.5-Math-7B, which is based on Qwen2.5-Math-7B, achieves the highest pass@1 score on the QHE benchmark.
- Furthermore, both Qwen2.5-Math-7B and AutoCoder_QW_7B achieve higher pass@1 scores compared to their respective
- baselines, demonstrating the effectiveness of our proposed method.

Generated Samples

```python
from qiskit import QuantumCircuit

def create_cz_gate():
    """ Design a CZ gate using only H and
    CNOT gates and return the quantum
    circuit.
    """
    qc = QuantumCircuit(2)
    # Step 1: Apply a Hadamard gate to the
    target qubit
    qc.h(1)
    # Step 2: Apply a CNOT gate with the
    control qubit as the control and the
    target qubit as the target
    qc.cx(0, 1)
    # Step 3: Apply a Hadamard gate to the
    target qubit again
    qc.h(1)
    # Step 4: Apply a CNOT gate with the
    control qubit as the control and the
    target qubit as the target
    qc.cx(0, 1)
    return qc
```

(a) Qwen2.5-Math-7B for problem 23 in QHE

```python
from qiskit import QuantumCircuit
def create_cz_gate()->QuantumCircuit:
    """ Design a CZ gate using only H
    and CNOT gates and return the
    quantum circuit.
    """
    qc = QuantumCircuit(2)
    qc.h(1)
    qc.cx(0, 1)
    qc.h(1)
    return qc
```

(a) qiskit-Qwen2.5-Math-7B for problem 23 in QHE

```python
from qiskit import QuantumCircuit
from qiskit import QuantumCircuit

def dj_constant_oracle() -> QuantumCircuit:
    """ Create a constant oracle for use in a
    Deutsch-Jozsa experiment. The oracle takes
    two input bits (qubits 0 and 1) and writes
    to one output bit (qubit 2).
    """
    # Create a quantum circuit with 3 qubits
    and 3 classical bits
    qc = QuantumCircuit(3, 3)
    # Apply a Hadamard gate to each qubit
    qc.h(0)
    qc.h(1)
    qc.h(2)
    # Apply a controlled-Z gate between qubits
    0 and 1
    qc.cz(0, 1)
    # Apply a controlled-X gate between qubits
    0 and 2
    qc.cx(0, 2)
    # Apply a controlled-X gate between qubits
    1 and 2
    qc.cx(1, 2)
    # Apply a Hadamard gate to qubit 2
    qc.h(2)
    # Measure qubits 0, 1, and 2
    qc.measure([0, 1, 2], [0, 1, 2])
    return qc
```

(c) Qwen2.5-Math-7B for problem 59 in QHE

```python
from qiskit import QuantumCircuit

def dj_constant_oracle():
    """ Create a constant oracle for use
    in a Deutsch-Jozsa experiment. The
    oracle takes two input bits (qubits
    0 and 1) and writes to one output
    bit (qubit 2).
    """
    qc = QuantumCircuit(3)
    # The oracle is constant, so we
    don't need to do anything special.
    return qc
```

(d) qiskit-Qwen2.5-Math-7B for problem 59 in QHE

We provide sample code generated by both Qwen2.5-Math-7B and qiskit-Qwen2.5-Math-7B. These two models are prompted with identical inputs, including the same import statements, function header, and Python docstring. While qiskit-Qwen2.5-Math-7B successfully generates a correct and executable quantum function, the original Qwen2.5-Math-7B fails to do so. This contrast illustrates the effectiveness of our fine-tuning approach.

Conclusion & Future Work

- In this work, we extend the code generation capabilities of large language models to the quantum domain, with a specific focus on Qiskit.
- Our experiment reveal a significant performance gap between general purpose code generation and quantum code generation.
- Our approach improves the model's ability to generate correct and meaningful quantum code.
- In future work, we aim to generate more accurate prompt-completion quantum code pairs to better finetune models and expand our efforts beyond Qiskit to include other quantum software frameworks.

References:

[1] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13, 2024.
[2] Andreas Köpf, Yannic Kilcher, Dimitri Von Rütte, Sotiris Anagnostidis, Zhi RuiTam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, RichárdNagyfi, et al. Openassistant conversations-democratizing large language modelalignment. Advances in neural information processing systems, 36:47669–47681,2023.
[3] Nicolas Dupuis, Luca Buratti, Sanjay Vishwakarma, Aitana Viudes Forrat, DavidKremer, Ismael Faro, Ruchir Puri, and Juan Cruz-Benito. Qiskit code assistant:Training llms for generating quantum computing code. In 2024 IEEE LLM AidedDesign Workshop (LAD), pages 1–4. IEEE, 2024.