

# Code Comprehension using Classical and Quantum CNN

Mobeen W. Alhalabi <sup>1</sup>, Fathy E. Eassa <sup>2</sup>

<sup>1</sup>Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University (KAU), Jeddah 21589, Saudi Arabia (e-mail: malhalabio005@stu.kau.edu.sa )

<sup>2</sup>Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University (KAU), Jeddah 21589, Saudi Arabia (e-mail: Feassa@kau.edu.sa)

---

## ARTICLE INFO

Received: 25 Dec 2024

Revised: 15 Feb 2025

Accepted: 25 Feb 2025

## ABSTRACT

Software developers often spend more time reading and understanding existing source code than writing new code. As a result, improving code comprehensibility can significantly enhance software development and maintenance by reducing the cognitive effort required to grasp program functionality. Code comments play a critical role in this process, helping developers navigate and maintain software more efficiently. However, numerous software projects suffer from missing, outdated, or inconsistent comments, forcing developers to infer functionality directly from the code. The design of programming languages also impacts code comprehension; as previous research suggests; different programming paradigms influence the ease with which developers understand code. This insight has driven researchers to explore machine learning (ML) and deep learning (DL) techniques for a variety of software engineering tasks, including testing, vulnerability detection, and source code analysis. Given the rapid growth in this research area, it has become increasingly difficult for the community to keep track of existing approaches and environments. Recently, ML and DL techniques have been widely adopted for tasks such as source code representation, quality analysis, and automated testing. In this paper, we introduce a novel quantum deep learning method that demonstrates superior performance over classical approaches in the comprehension of source code. Our approach highlights the potential of quantum computing to enhance machine learning techniques applied in the software engineering domain.

**Keywords:** Code Comprehension, Convolutional Neural Network, Deep Learning, Java Source Code, Natural Language Processing, Quantum Neural Network.

---

## I. INTRODUCTION

THE high energy physics (HEP) community has a long-standing tradition of managing extensive datasets and employing sophisticated statistical methods to examine experimental data across the energy, intensity, and cosmic frontiers. The HEP community requires substantial computational advancement to manage escalating data quantities, and quantum information science (QIS) innovations may offer a feasible answer. Quantum supremacy refers to the ability to solve problems more rapidly than any classical approach. In the context of computational complexity theory, this often means delivering super polynomial acceleration compared to the most efficient known classical method. In this study, we will define quantum advantage as the superiority of quantum devices over

classical ones, not necessarily characterized by super polynomial speedup. Machine learning (ML) techniques provide significant advantages for scalable data analysis. The surge in deep learning (DL) development originates from recent progress in convolutional neural networks (CNNs). In conjunction with big data and graphics processing unit (GPU) capabilities, DL has markedly enhanced the capacity to evaluate extensive quantities of text data. Numerous instances exist in which CNNs have been effectively utilized to address HEP difficulties by employing classical computers [1].

Nevertheless, substantial advances in the implementation of robust representation learning algorithms in quantum computing have not yet been achieved. Nevertheless, it is crucial to investigate the potential of quantum ML, which has lately garnered significant attention. This research introduces a novel hybrid quantum convolutional neural network (QCNN) framework to illustrate the quantum advantage over equivalent conventional techniques. We evaluated its efficacy in classifying high-energy physics events using simulated data from neutrino investigations. We show that for a comparable number of parameters in the QCNN and traditional CNNs, the QCNN can achieve faster learning or achieve superior testing accuracy with fewer training epochs. Our simulations indicate a potential empirical quantum advantage of QCNNs over CNNs with respect to the accuracy of the testing [1].

Our classical knowledge derives from our life experiences; however, these experiences do not encapsulate the fundamental mechanisms of nature. Quantum mechanics, the fundamental principles that govern our reality, account for all observable phenomena. The phenomena of quantum mechanics are incongruent with conventional reasoning. Quantum computing employs the principles of quantum mechanics to perform computational tasks; quantum information refers to the use of quantum phenomena to enhance information transfer, storage, and retrieval, including the development of sophisticated cryptographic techniques. The many permutations associated with quantum processing enable quantum computers to double their memory capacity with the addition of each qubit, which is the source of quantum computing's power. Quantum computers can solve problems unattainable by classical computers, exemplified by their capacity to factor enormous numbers by the Shor algorithm. Moreover, in certain instances, the integration of classical and quantum computers might yield enhanced problem solving efficiency, with quantum algorithms demonstrating superiority over their classical counterparts [2].

Quantum ML involves using the potential of quantum computers to develop scalable machine learning models that surpass the performance of conventional models while being more cost-effective [3]. The principles of quantum computing (QC) investigate the challenges associated with data storage, processing, and analysis [4]. Quantum mechanical systems are established by transforming information, consistently designated quantum information. Quantum information denotes data that represent the state of a quantum system [5]. The fundamental concept of quantum information pertains to the state of any quantum system characterized by two distinguishable degrees of freedom; the logical values 0 and 1 are referred to as a Qubit. A quantum computer exhibits counterintuitive phenomena of quantum physics, such as superposition, entanglement, and tunneling. Consequently, a quantum computer can quickly address issues that exceed the capabilities of classical devices [6].

Quantum computing is a rapidly advancing field that utilizes the principles of quantum mechanics to tackle complex problems that pose challenges to classical computing. Machine learning is a specialized area within artificial intelligence that enables computers to identify patterns based on their experiences. Rapid increase in data volume presents challenges for traditional machine learning algorithms in handling big data, while quantum computing offers the potential for significantly faster processing capabilities. The integration of quantum computing with machine learning has led to the emergence of a novel domain called quantum machine learning. Quantum machine learning algorithms leverage the rapid processing capabilities of quantum computing, demonstrating a significant speed advantage over classical methods.

Natural language processing (NLP) represents a significant domain within artificial intelligence, facilitating the computer's comprehension of human languages. Currently, efforts are underway to take advantage of the speed advances of quantum machine learning in applications related to natural language processing [7]. Various techniques and their adaptations have been developed to address tasks in understanding source code, including code representation and other downstream applications. Although these approaches have achieved notable progress, significant challenges remain in structural code comprehension, which can be summarized as follows [8]:

- **Structural Modeling of Code:** Traditional language models typically process source code as sequential token inputs, often overlooking the inherent structural information. Consequently, critical challenges emerge, such as how to effectively model and leverage structural features within code and how to select the most relevant structural information for specific downstream tasks.
- **Learning Generic Code Representations:** Much of the existing research is dedicated to learning representations for individual programming languages, making it difficult to develop models that capture language-agnostic code features. Addressing this issue involves discovering methods to learn programming language-independent code representations that overcome language-specific limitations.
- **Task-specific Code Adaptation:** Adapting models for downstream tasks still presents considerable hurdles, including how to design and select appropriate architectures for tasks like code generation and program repair, how to preprocess datasets for specific task requirements, and how to adapt models in settings such as few-shot learning, transfer learning, and cross-language applications.

The remainder of this paper is organized as follows. Section II discusses the literature review of previous related work; Section III introduces the data used in this paper, Section IV shows the performance of CNN and QCNN on the experimental data used, and Section V is focused on results and discussion, Section VI comparative analysis of the results, and VII summarize our conclusion.

## II. LITERATURE REVIEW

Our study will be compared with other research based on the three phases of similarities: the type of dataset, the different algorithms used to solve the same problem and the QCNN algorithm. The first study [9] employed two orthogonal methodologies for the task: information retrieval (IR) and neural-based techniques, using DeepCom [10] and a Cross-Encoder-based classifier [11] on the FunCom dataset. The evaluation was carried out on a large-scale data set of Java applications. The experimental results indicate that the method achieves a BLEU score of 25.45, outperforming the state-of-the-art information retrieval technique, the neural based approach and their amalgamation by 41%, 26%, and 7%, respectively.

On the other hand, [10] also used the DeepCom dataset; the study used Hybrid-DeepCom and a variation of the attention-based Seq2Seq model to create comments for Java methods. The results assess the effect of each suggested modification in the proposed methodology. The sentence-level BLEU score increases from 27.88% to 34.57% utilizing the neural machine translation model, such as the Seq2Seq model. The enhancement indicates that framing the comment-generating issue as a machine translation effort significantly influences performance. The attention mechanism improves the Seq2Seq model by 3%. Hybrid-DeepCom can produce more insightful comments by including structural information. Compared to the model devoid of Abstract Syntax Tree (AST), namely Seq2Seq (Attention), the Hybrid-DeepCom BLEU score exhibits an increase of 11%. Compared to the AST-based model DeepCom, Hybrid-DeepCom exhibits an increase of 1.3%. The BLEU score rose to 39.51% with the application of Beam Search during comment generation. The BLEU score increases to 41.86% using camel case splitting. Experimental findings suggest that the comment generation task closely resembles machine translation. The acquisition of lexical and structural information and the segmentation of camel cases enhance the generation of code comments.

[12] presents a novel neural network-based technique, SeCNN, to produce code comments for Java methods. SeCNN employs CNN to mitigate the long-dependency issue in source code manipulation and incorporates numerous innovative components to capture the semantic information of the source code. Specifically, they develop a CNN component based on source code to acquire lexical information and an AST-based component to obtain syntactical information. Subsequently, they employ LSTM integrated with an attention mechanism to decode and produce code comments. DeepCom performs extensive tests, revealing that SeCNN outperforms five leading strategies, with a BLUE score of 44-44.89%. Compared to analogous baselines, Hybrid-DeepCom and AST-attendgru, which similarly utilize source code and AST, SeCNN has superior efficiency performance, as evidenced by its significantly lower execution time.

The investigation by [13] on the explainability of large language models in code summarization used SHAP (SHapley Additive exPlanations) with the FunCom dataset. In light of the vast research in natural language processing that suggests that aligning neural models with human visual patterns improves performance, we restrict our conclusion to the SHAP measure of feature attribution and human attention as evaluated in an eye-tracking experiment. Much research is being conducted to investigate the connection between human attention and the attribution of features in computer models. Their research demonstrated that SHAP did not match the visual attention of humans in the measurements or models investigated. The findings indicate that the use of SHAP for feature attribution does not enhance the ability to explain language models by establishing correlations between machine and human focal points. Feature attribution assessed by SHAP may not be the most effective method for interpreting a language model's emphasis during code summarization, as it does not align with human attention patterns. As an alternative, it could be that machines approach the reasoning of code in a manner distinct from humans when assigned the task of summarizing source code.

After finding studies assessing code comprehension from a human developer's perspective, [14] constructed an aggregated data set that included information from ten research projects that included over 427 code samples and about 24,000 unique human evaluations. With the help of these data, an exhaustive meta-analysis was carried out to evaluate the relationship between Cognitive Complexity and several measures of source code understandability. Within the scope of their work, the authors validated a metric known as Cognitive Complexity. This metric was developed specifically to evaluate the understandability of code, and it has already gained widespread adoption due to its incorporation into established static code analysis tools. An exhaustive search of the relevant literature was performed to collect data sets from research investigating the understandability of code, resulting in almost 24,000 reviews of the understandability of 427 code samples.

A meta-analysis allowed a statistical summarization of the correlation coefficients and quantification of the correlations between these measurements and the associated metric values. A positive correlation exists between cognitive complexity, the amount of time required for comprehension, and subjective judgments of understandability. There was a lack of consistency in the findings concerning the connection between the metric and the accuracy of comprehension tests and physiological indicators. Overall, this research illustrated that code comprehension can be measured in various ways, and how these approaches are related is not known.

Question Type Classification Methods Comparison was one of the models evaluated by [15]. All models were trained and evaluated using the TREC 10 dataset; the CNN-based technique, utilizing kernel sizes ranging from 2 to 6 and a single fully connected layer, demonstrated the highest performance, achieving an accuracy of 90.7%.

For the work in [16], classical CNN and NN models attained accuracy ratings of 0.999 and 0.9141, respectively. However, the quantum QNN and QCNN models exhibited accuracy scores ranging from 0.5 to 0.6 and 0.52 to 0.61, respectively. The limited scale of the MNIST dataset, comprising merely 60,000 training images and 10,000 test images, which, following necessary preprocessing, is

diminished to 1,000-2,000 training and testing images, respectively, is likely a contributing factor to the inferior performance of QCNN algorithms in binary classification tasks relative to traditional CNN and NN. A potential reason for the suboptimal performance of QCNN algorithms is their relatively novel and intricate architecture, which may necessitate increased optimization efforts.

Furthermore, the hardware constraints of quantum computers, including their restricted qubit count and coherence durations, may contribute to their diminished efficacy on tiny datasets such as MNIST. Furthermore, the susceptibility of quantum computers to noise and mistakes is an additional aspect that may influence the accuracy and performance of QCNN algorithms, especially in the context of near-term quantum computers characterized by elevated error rates. Ultimately, the MNIST dataset may not provide a distinct quantum advantage over conventional algorithms, suggesting that the efficacy of QCNN algorithms may not much exceed that of traditional CNNs and NNs in this instance.

Another application of QCNN is the classification of high-energy physics occurrences [17]. The suggested model is evaluated using a simulated Deep Underground Neutrino Experiment dataset. The suggested quantum architecture exhibits a superior learning speed compared to classical CNNs while maintaining a comparable number of parameters. The QCNN converges more rapidly and attains superior test accuracy relative to CNNs. The numerical simulations indicate that applying QCNN and other quantum machine learning models to high-energy physics and various scientific domains is a promising avenue. The results demonstrated accuracy rates for different binary classification metrics on the testing dataset: CNN achieved 80%, 82%, and 95%, while QCNN

attained 92.5%, 97.5%, and 98.5%.

In [18], one of the most promising quantum machine learning (QML) designs, QCNN, was adopted. The authors examined the classification of quantum phases of matter, a cutting-edge application. The architecture of the QCNN, with its equivariant and pooling layers, yields an ansatz with limited expressivity. Fundamental attributes, such as intermediate measurements, parameter sharing, and logarithmic depth, enhance bias while reducing variation. This indicates that QCNN is expected to have superior generalization capabilities compared to conventional hardware-efficient approaches. Most complexity measures are monotonically related to the expressivity of the function family, and uniform generalization bounds are monotonically related to a complexity measure. Consequently, the argument that uniform generalization limitations for the QCNN family are trivially loose indicates that the same bounds applied to less constrained models must likewise be vacuous.

In this regard, the findings for QCNNs extend to the complete domain of unrestricted QML. It was analytically demonstrated that polynomially sized quantum NNs may accommodate any arbitrary labeling of data sets. This appears to counter the assertions that limited training data are demonstrably inadequate to ensure effective generalization in QML. Finally, analytical and numerical findings do not exclude the potential for effective generalization with limited training data; nonetheless, they suggest that they cannot ensure this based on uniform generalization boundaries. The factors contributing to the successful generalization remain unidentified.

### III. METHODOLOGY

We proposed two models for the same dataset, CNN and QCNN, to evaluate the code comprehension task; each model gave high-accuracy classification results as described below. For the CNN model, the accuracy achieved was 96%, while QCNN gave an accuracy of 98%. The proposed strategy is organized into 4 phases:

- 1) Data preprocessing,
- 2) Tokenizer,
- 3) CNN algorithm,

## 4) QCNN algorithm.

## A. DATA PREPROCESSING

The code delineates a function called `create_dataset` that takes two file paths as arguments: one for code and another for NL comments. The function examines these files and prepares them for subsequent use. Subsequently, the data preprocessing phase accepts the file path within the function, which encompasses variables stored in paths to the input files: One file contains code, while the other provides the equivalent NL comments. The subsequent process in that data preprocessing phase involves defining the function `creat_dataset`, which accepts two parameters: a: the path to the code file and b: the path to the natural language file. Perusal The Code File follows in this step, which opens the supplied code file and reads its lines into a list named `lines1`, with each line in `lines1` being handled as a: The newline character (`\n`) is removed after each line. Each line is enclosed with tokens that signify the beginning and end of each code snippet.

Subsequent reading of the Natural Language File is as follows: Analogously to the preceding block, this code accesses the NL comment file designated by b and imports its lines into a list named `lines2`. Each line undergoes identical processing, encapsulated with and tokens. Subsequently, the length of the data is verified. After processing both files, the program verifies whether the quantity of code snippets (lines in `lines1`) corresponds to the quantity of natural language comments (`lines2`). If the counts are incongruent, a warning message is generated. A Return Statement is established to return two lists: `lines1`, which contain the processed code snippets. And `lines2`: The natural language remarks analyzed. The `creat_dataset` utility efficiently prepares a dataset by reading code and natural language comment files. Enclose each line with tokens to enhance processing in subsequent tasks, such as building a sequence-to-sequence model. Verify the consistency of code snippets and comments before returning the processed lists.

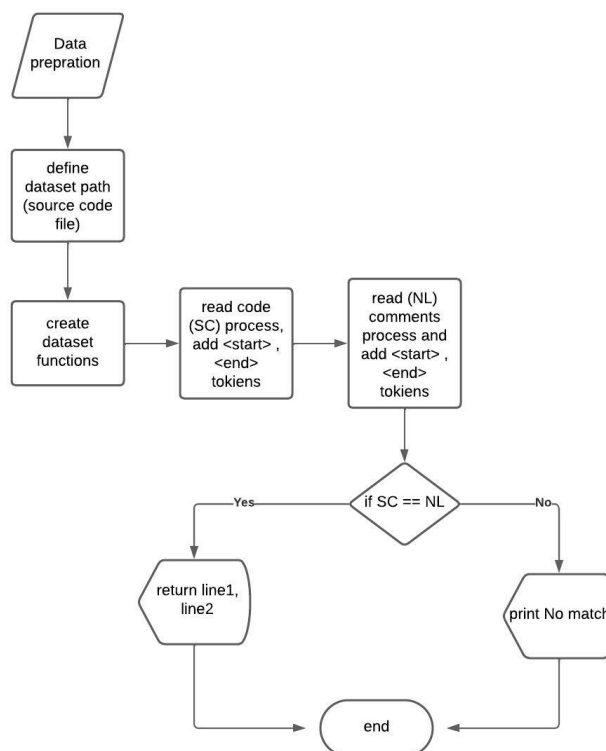
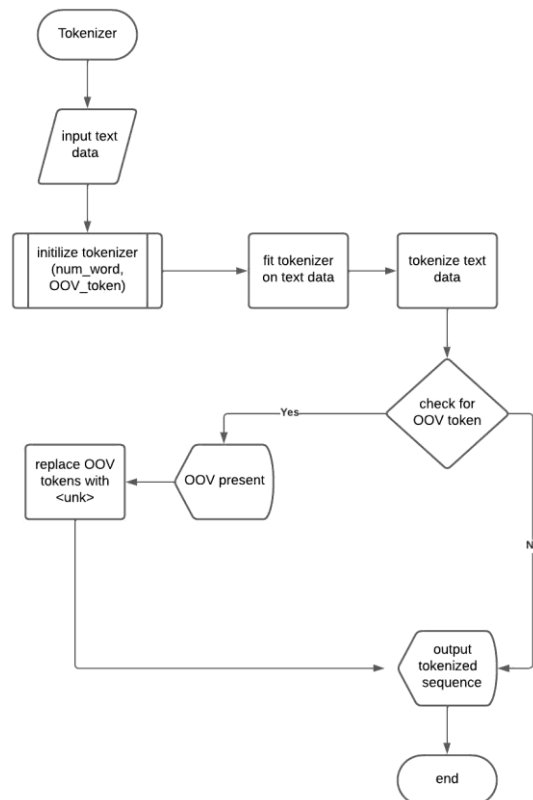


FIGURE 1. Data preprocessing process

This is generally required by our machine learning tasks with respect to code summary and comment generation. This process concludes in **FIGURE 1**, which is the flow diagram.

### B. TOKENIZER

This work uses TensorFlow's `tf.keras.preprocessing.text`; it is a function that tokenizes text data by processing the text, constraining the vocabulary, generating a mapping table, and substituting unknown words. Initial segment Establish Constants: `code_maxlen = 150`; Maximum length for code snippet- pets (150 tokens); `nl_maxlen = 30` The maximum length for natural language comments is 30 tokens, with a vocabulary limit of 30,000. The vocabulary's maximum size is 30,000 words.



**FIGURE 2.** Tokenizer process

After that, the tokenization function is defined as *def tokenize (lang, maxlen)*, where *lang* represents the textual material designated for tokenization (which may include code or natural language), and *maxlen* is the upper limit for the sequence lengths post-tokenization. The subsequent step in the process involves the creation of a tokenizer using `tf.keras.preprocessing.text.Tokenizer` by instantiating a tokenizer with the following parameters: i) *num\_words* set to `vocab_maxlen + 1`. It restricts the size of the vocabulary to 30,001 words (inclusive of the OOV (out of vocabulary) token

), ii) *filters* set to `filters=""`. This implies no character filtering is possible (normal behavior would exclude punctuation and special characters); iii) *oov\_token* is set to "unk", designating a token for out-of-vocabulary words (words absent from the vocabulary). Subsequently, the tokenizer is fitted to the text data; this approach analyzes the input text (*lang*) and constructs the vocabulary according to word frequency. Each distinct word is assigned a specific index. The text input is transformed into sequences of integers, with each integer representing a word's index in the lexicon. For instance, "hello" could be denoted as five if it is the fifth entry in the lexicon.

Furthermore, the Pad and Truncate sequences are applied as follows: i) *pad\_sequences* guarantee

uniformity in sequence length; ii) *maxlen* enforces the maximum length defined in the function call; iii) *padding* is set to 'post'. This appends padding to the end of the sequences; iv) *truncating* is set to 'post'. This removes excess sequences from the end when they exceed the maximum length. The concluding function is the Return Statement, which yields a tensor. The tokenized sequences are padded and truncated using `tf.keras.preprocessing.sequence.pad_sequences`. This ensures that all sequences are of uniform length (`code_maxlen` for the code and `nl_maxlen` for the natural language). Padding is added at the end (post padding), and if a sequence exceeds the defined length, it is truncated. The tokenize function is intended to preprocess textual material for application in ML models as follows:

- 1) Initialize a tokenizer that constrains the vocabulary and specifies an out-of-vocabulary token.
- 2) Trains the tokenizer on the supplied text data to establish a vocabulary. Transforms the text into integer sequences according to the vocabulary.
- 3) Pads or truncate these sequences to achieve uniform length.

This Is Important for Code Comprehension for the following reasons:

- The code and the comments are both text, but with different patterns and structures.
- Tokenization:
  - Transforms them into machine-readable input.
  - Ensures vocabulary is manageable.
  - Handles unknown or rare words consistently.
- After tokenization, the data are usually:
  - Padded to a uniform length (important for batching).
  - Passed to deep learning models (like LSTMs, Transformers, or BERT-based models) to perform tasks such as summarizing code, identifying bugs, or generating documentation.

This tokenizer process prepares the code and the NL text for deep learning by:

- Creating a vocabulary,
- Mapping text to sequences of integers,
- Handling unknown words,
- Outputting clean, uniform sequences.

It is a critical preprocessing step before any model can start learning patterns in code comprehension tasks.

It is a formatting text data for deep learning models, which necessitates fixed-size input. **FIGURE 2** graphically represents this process as a flow diagram. In short, it does the following processing steps:

- 1) Input Text Data: Raw code or natural language (NL) text is provided as input.
- 2) Initialize Tokenizer:
  - Using `tf.keras.preprocessing.text.Tokenizer`.
  - Set a vocabulary limit (for example, 30,000 words).
  - Define an OOV token ("unk") to handle unseen words.
- 3) Fit Tokenizer on Text

- The tokenizer learns the vocabulary by analyzing the input text.
  - Each unique word is assigned an index based on frequency.
- 4) Tokenize Text Data: The input text is converted into sequences of integers.
  - 5) Check for OOV Tokens: The tokenizer checks if there are any out-of-vocabulary (OOV) words.
  - 6) Handle OOV tokens (if present): Replace them with the OOV token (unk) to maintain consistency.
  - 7) Output Tokenized Sequence: The final result is a numerical representation of the input text ready for the model.

### C. CNN ALGORITHM

The Convolutional Neural Network (CNN) is a type of feed-forward neural network capable of automatically extracting features from input data through convolutional operations. Unlike traditional feature extraction techniques, CNNs eliminate the need for manual feature engineering. The design of CNNs is inspired by human visual perception: artificial neurons mimic biological neurons, convolutional kernels act as specialized receptors that detect distinct features, and activation functions replicate the biological mechanism where only signals that exceed a threshold are propagated to subsequent neurons [21].

Loss functions and optimization algorithms are employed to guide the CNN learning process toward the desired outcomes. Compared to fully connected (FC) networks, CNNs offer several key advantages: (1) Local connectivity, each neuron connects only to a limited number of neurons in the previous layer, reducing the number of parameters and accelerating convergence; (2) Weight sharing multiple connections share the same weights, further minimizing the model's complexity; (3) Dimensionality reduction via down sampling pooling layers exploit local correlations within data to down-sample inputs, retaining essential information while discarding redundant features. This not only decreases data volume, but also reduces the number of trainable parameters. Together, these properties make CNNs one of the most influential and widely adopted architectures in deep learning [22]. The proposed architecture for the CNN model **FIGURE 3** used in this work is as follows:

#### 1) Inputs:

- Code Input (encoder\_input):

The inputs are represented by: i) Code Input (encoder\_input): A sequence of code tokens submitted to the encoder. This is structured as (code\_maxlen, 1), signifying a sequence of length code\_maxlen with a single channel, perhaps representing one-dimensional tokenized code; and ii) Natural Language Input (decoder\_input): A series of natural language tokens (comments) supplied as input to the decoder. It is configured as (nl\_maxlen,).

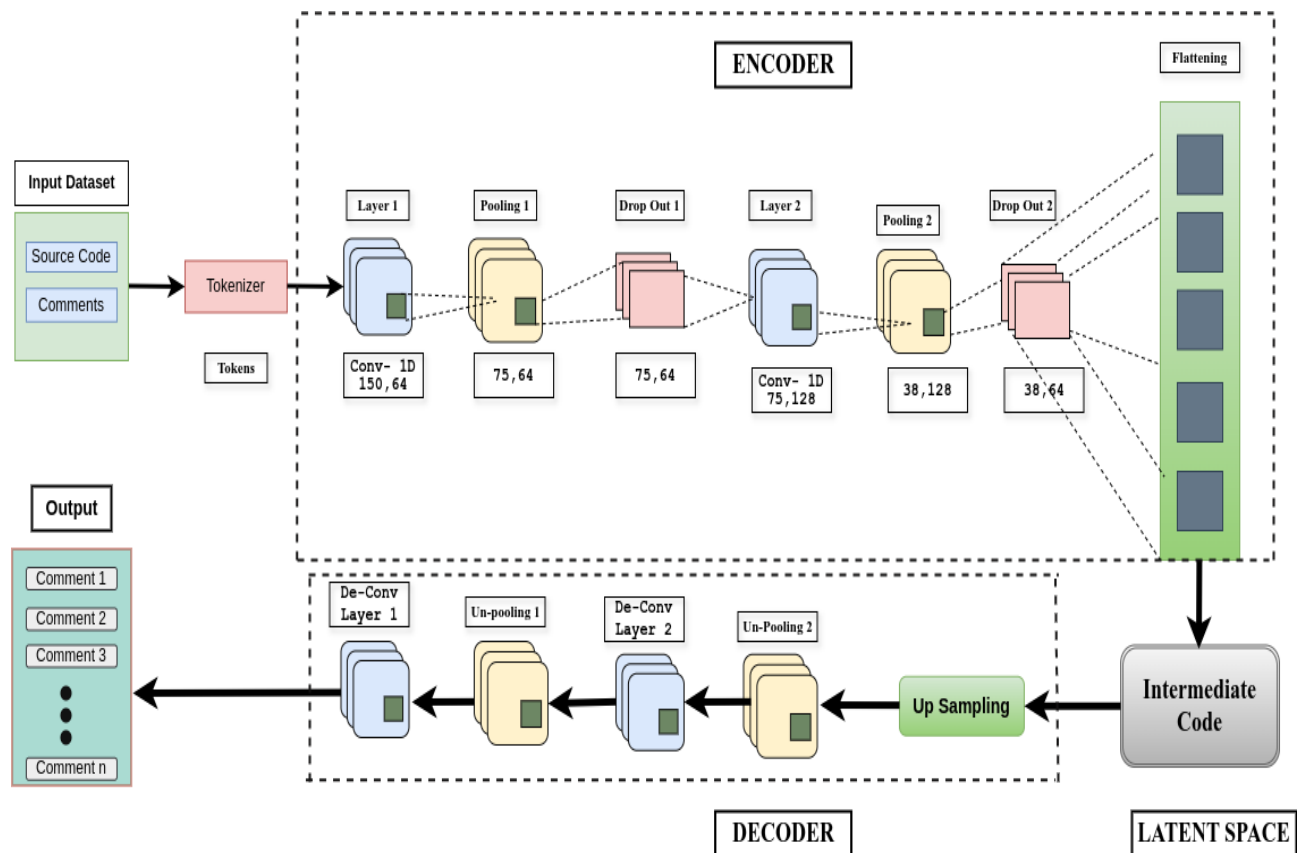


FIGURE 3. CNN model architecture

- Natural Language Input (decoder\_input): input to the de- coder as a series of natural language tokens (comments).

<nl\_maxlen,> is its configuration.

## 2) Encoder (CNN):

- The coding sequence is analyzed using a 1D CNN:
  - Conv1D Layer 1: Uses 64 filters with a kernel width of 3 tokens and uses ReLU activation and padding to keep the input dimensions.
  - MaxPooling Layer 1: The sequence length is re- duced.
  - Conv1D Layer 2: 128 filters of an analogous kernel size are utilized.
  - MaxPooling Layer 2: It further reduces the se- quence length by fifty percent.
  - Conv1D Layer 3: Employs 256 filters.
  - MaxPooling Layer 3: Froker further compresses the sequence, keeping the 3D tensor configuration intact. It has an output of the encoder as a 3D tensor with the shape (batch\_size, reduced\_steps, 256), re- duced\_steps signify the sequence length after three pooling operations. With a single channel, perhaps representing one-dimensional tokenized code.
- The output of the encoder is a 3D tensor with dimensions (batch\_size, reduced\_steps, 256), where reduced\_steps represent the sequence length following three pooling operations.

## 3) Decoder(LSTM):

The natural language input (a series of tokenized remarks) is processed through:

- **Embedding Layer:** Does embedding lookup, or 'embedding,' of the input tokens into dense (vector) embeddings of dimension embedding\_dim.
- **LSTM layer:** Instead of just an end state as output, the algorithm sequentially processes the embedded tokens and outputs the complete series of hidden states.

## 4) Attention Mechanism:

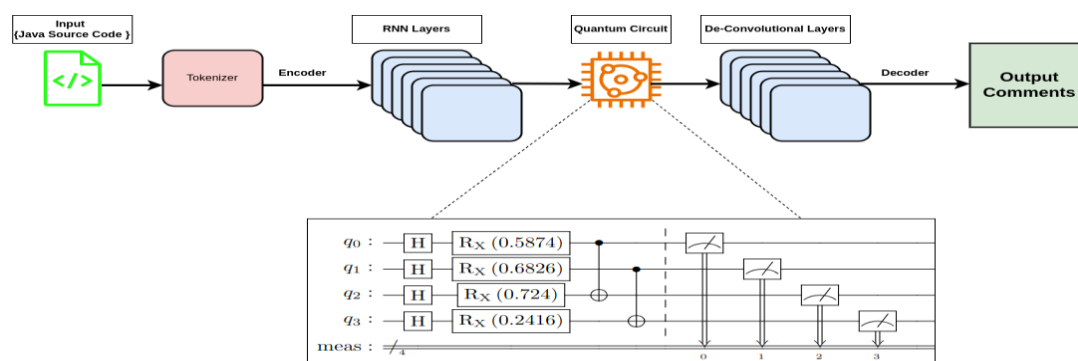
In the output of the LSTM decoder lies the encode code sequence. During the generation of each output token, the attention mechanism calculates a weighted sum of the output of the encoder to effectively highlight the most relevant parts of the input sequence.

## 5) Output Layer:

A Dense layer with vocab size units and softmax activation is used to transmit the output using vocab\_size units, forecasting a probability distribution over the vocabulary for each token in the output sequence.

## D. QCNN ALGORITHM

Quantum computing investigates quantum mechanics, a discipline of physics that elucidates the behavior of matter and energy on the microscale [23]. The architecture proposed for the QCNN model in **FIGURE 4** used in this work is as follows:



**FIGURE 4.** QCNN model architecture

## 1) Inputs Layer:

- **Encoder Input:** For the source code, Shape (code\_maxlen, 1).
- **Decoder Input:** Natural language output of shape (nl\_maxlen,)

## 2) Encoder (CNN):

- **Conv1D Layer:** Kernel Size 3, ReLU Activation and 64 filters with L2 regularization.
- **MaxPooling1D Layer:** Pool size 2.
- **Dropout layer:** 30% dropout to prevent overfitting.
- **Conv1D Layer:** Kernel size 3, ReLU activation, L2 regularization, output 128 filters.
- **MaxPooling1D Layer:** Pool size 2.

- Dropout Layer: In this experiment, we also added a dropout layer with dropout rate of 0.3 to the QCNN as we observed dropout 30%. We see that the dropout operation greatly reduces the problem of overfitting in this case.
- Conv1D Layer: A ReLU activation layer is used, with L2 regularization and 256 filters with a kernel size of 3.
- MaxPooling1D Layer: Pool size 2
- Flatten Layer: Flattens and outputs so that it can go into the decoder.
- RepeatVector Layer: It has to repeat the encoder output to fill the decoder sequence length.

3) Latent Representation:

Dense Layer: Outputs a latent vector of size latent\_dim (4).

Quantum Layer:

Integrates the quantum circuit output, which processes the latent representation.

4) Decoder (CNN):

- Embedding Layer: This takes a token of any index and maps it to a dense vector of dimensionality embed\_dim.
- Reshape layer: Metrics adjusted for CNN processing.
- Conv1D Layer: It is 256 filters, kernel 3, ReLU activation.
- Conv1D Layer: Filters 128, kernel size 3, ReLU activation.
- Concatenate Layer: The output of a quantum layer is merged with the output of the decoder.
- Conv1D Layer: A balanced feature extractor of 64 filters, kernel size 3, ReLU activation.

5) Output Layer:

Dense Layer: Softmax activation is used to produce predictions for vocabulary size.

6) Quantum Circuit Simulation

The figure above **FIGURE 4** is a simulation of a quantum circuit used in this experiment; below is an explanation of this simulation.

1) Qubits

We have 4 quantum qubits labeled q0 through q3, and one classical register (meas) with 4 bits to store the measurement results.

2) Gates Applied

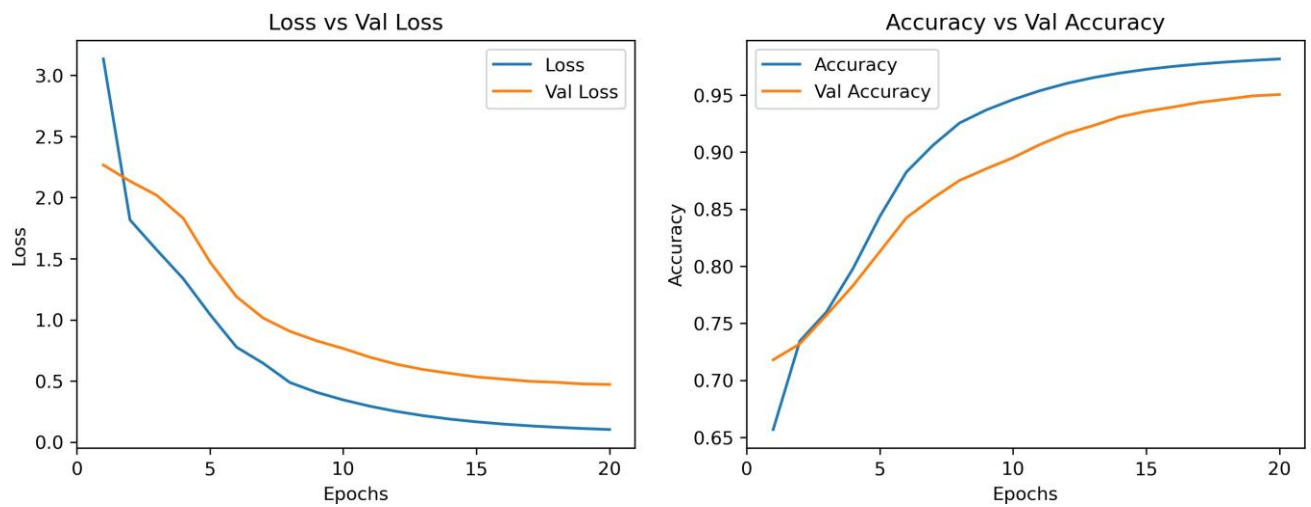


FIGURE 5. CNN Model Loss and Accuracy

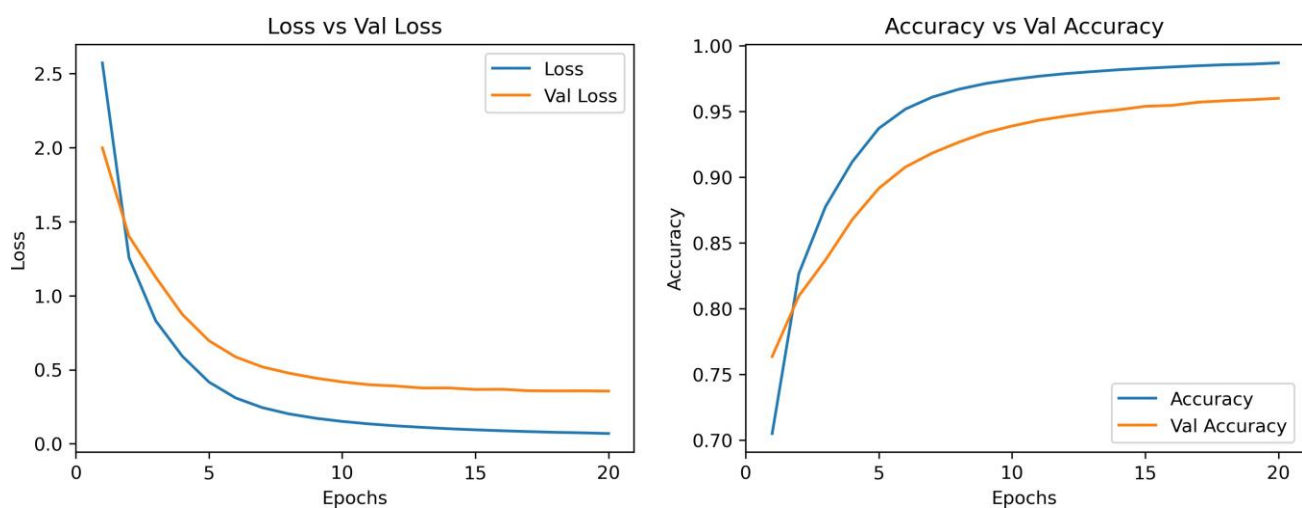


FIGURE 6. QCNN Model Loss and Accuracy

- Hadamard Gates (H)

Each qubit starts with a Hadamard gate, which puts them into a superposition [24]:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad (1)$$

This prepares each qubit to interfere with others in the Quantum Fourier Transfer (QFT) process.

- Rotation Gates (Rx)

Each qubit is acted upon by a rotation around the X-axis,  $R_X(\theta)$ , where  $\theta$  is the angle given on each value [24]:

-- q0:  $R_X(0.969)$

-- q1:  $R_X(0.909)$

-- q2:  $R_X(0.282)$

-- q3: Rx(0.358)

Controlled Operations

There are two controlled-addition (or controlled- Rx) gates:

-- One between q0 (control) and q2 (target)

-- The other is between q1 (control) and q3 (target)

This run on local machine using Accelerated Emulation Resources (AER) Simulator. We can also run this model on Real IBM Quantum Computer, which has one free account and the other paid account [25].

#### 7) Limitations of current quantum hardware

IBM has developed a range of quantum processors, each with varying qubit counts and coherence times [26], we used the Eagle Processor in our evaluation. **TABLE 1** represent an overview of some notable IBM quantum processors:

#### Decoherence Times ( $T_1$ and $T_2$ ):

Specifically  $T_1$  (relaxation time) and  $T_2$  (dephasing time), are critical metrics that indicate how long a qubit can maintain its quantum state. For IBM's superconducting qubits, these times typically range between 50 to 100 microseconds, though exact values can vary depending on the specific processor and qubit. IBM has implemented techniques such as TLS (Two-Level System) mitigation to improve coherence times across its processors [27].

Consequently, numerous quantum programming languages have been introduced to facilitate the specification of quantum algorithms. Quantum algorithms often include the conversion of quantum circuits into code, namely a series of textual programming statements [28]. The encoder analyzes the input code using a wide range of convolutional layers to extract information. A latent representation is produced and transmitted across a quantum layer. The decoder, initialized with the natural language input, reconstructs the output sequence utilizing its convolutional layers. The ultimate output is generated through a thick layer that forecasts the subsequent token in the sequence.

#### E. DESIGN DECISION METRICS

Each convolutional filter slides across 3 tokens at a time in the input sequence.

#### Why kernel size = 3 effective?

##### 1) Local Context Capture:

Kernel size 3 captures trigram-level features — groups of 3 adjacent tokens or signal values. This is especially useful in natural language (e.g., capturing "not good", "very fast") and time series where local dependencies matter.

##### 2) Shallow but Powerful:

Smaller kernels allow deeper networks, stacking multiple layers to capture hierarchical features (like edges → textures → shapes in images; or phrases → sentences in NLP).

##### 3) Balance Between Simplicity and Expressiveness: Larger kernels can overfit or become too parameter-heavy. Size 3 is a sweet spot: small enough to generalize, large enough to capture meaningful patterns.

##### 4) Empirically Proven:

Various architectures (e.g., VGG in image models, or character-level CNNs for text) use kernel size = 3 because it consistently performs well [27].

### **What L2 Regularization Does?**

Penalizes large weights by adding a term to the loss [29]:

Prevents Overfitting:

- Especially critical in the model since CNNs can easily memorize patterns.
- L2 promotes weight decay, forcing the model to generalize better to unseen data.

#### 2) Smoothens Learning:

Encourages smaller, smoother weights, which helps in stabilizing training and avoiding overly complex solutions.

#### 3) Acts as a Regularizer in Dropout+CNN/QCNN combo: we are using Dropout(0.3) — pairing it with L2 helps regularize both neuron activation (dropout) and weight magnitude (L2).

#### 4) Improves Generalization in Low-Data Regimes:

If we are working with a relatively small dataset (common in quantum-NLP hybrids), L2 is a simple way to keep the model grounded.

### **Filter Size**

The filter sizes we are referring to: 64, 128, and 256 are part of a progressively increasing convolutional architecture, which is a common and powerful design pattern in deep learning, especially for sequence and image processing.

Each number (64, 128, 256) is the number of filters (a.k.a. kernels or feature detectors) applied at that layer. Each filter is a learnable weight matrix that slides across the input to detect a specific pattern.

### **Why Increasing Filter Sizes:**

#### 1) Hierarchical Feature Extraction:

- Early layers with 64 filters capture simple, low-level features (e.g., edges, basic word patterns).
- Deeper layers with 128 or 256 filters learn more complex and abstract patterns, formed by combining simpler features.
- This progression mimics human perception: simple to complex.

#### 2) Wider Layers Compensate for Reduced Size:

As we apply MaxPooling, the spatial (or temporal) dimension shrinks, increasing the depth (filters) to preserve and enhance the representational capacity of the model.

#### 3) Better Generalization in Deep Networks: More filters in deeper layers means:

- More expressive power
- Richer representations
- Ability to learn diverse features

This enables the model to capture subtleties in input sequences useful for tasks such as code summarization or text generation.

$$\text{Loss} = \text{Original Loss} + \lambda \sum w_i^2$$

(2)

Why These Specific Numbers (64 → 128 → 256)?

This follows a power-of-2 progression (see **Table 2**):

This discourages the model from relying too heavily on any one feature or neuron.

### Why L2 is Important?

#### Easy to parallelize on GPUs

- Fits well into memory blocks
- Widely used in standard CNNs like VGG, ResNet, and Transformers

**TABLE 1.** IBM quantum processors types

Process or	Qubits	Notable Features
Heron R2	156	Heavy-hexagonal lattice; improved coherence via TLS mitigation
Heron R1	133	First-generation Heron processor with enhanced stability
Osprey	433	Nearly quadruple the size of Eagle; designed for scalability
Eagle	127	Introduced advanced control systems for improved performance
Falcon	27-65	Early generation processors with foundational architectures

It's not a rule, but it's a strong empirical convention.

**TABLE 2.** Filter size used in the experiment design

Filter Size	Purpose
64	Captures basic patterns (low-level features)
128	Captures intermediate patterns (phrases)
256	Captures high-level patterns (semantics, code blocks)

### Dropout rate:

The model uses a dropout rate of 0.3 (30%) to avoid overfitting. This value can be adjusted based on the complexity of the data set and the tendency of the model to overfit - high dropout for more complex models or severe overfitting, and lower dropout for simpler data sets or under fitting scenarios.

With quantum layers, gradients may be fragile → smaller learning rates are better. and if there is an overfitting; it can be

hadeled by increasing dropout or using early stopping. where if underfitting the objects, it can be handeled by increasing units, filters, or embedding dimension.

### QCNN training process:

We are using

- Optimizer: Adam
- Learning Rate: 0.0001

### Why Adam?

Adam (Adaptive Moment Estimation) combines the best of:

- Momentum (smooths gradients)
- RMSprop (scales learning rate adaptively)
- It is particularly well-suited for handling sparse gradients and noisy datasets, which are common characteristics in NLP and code comprehension tasks.
- Requires minimal tuning and converges fast.
- Loss: Sparse Categorical Crossentropy

Due to output layer is layers.Dense(vocabsize, activa- tion=softmax) which predicts a probability distribution over vocabulary. Labels (target sequences) are assumed to be Integer-encoded (not one-hot) hence sparse ver- sion is used; This loss [29]:

$$\text{Loss} = -\log(P(y_{\text{true}} | x)) \quad (3)$$

Penalizes the model based on how close the predicted distribution is to the true label.

**Hyperparameters: How Are They Chosen or Tuned?** TABLE 3 conclude our hyperpaprameter used in our investi- gation:

Grid search / Random search - Test multiple combina- tions.

- 1) Bayesian Optimization - Smarter exploration of hyper- parameter space (e.g., via Optuna or Keras Tuner).
- 2) Manual tuning + intuition - especially in quantum-NLP models, where pre-trained results are rare.

Manual settings; could benefit from tuning tools such as Keras Tuner or Optuna [30].

Quantum computer programming often involves the estab- lishment of a quantum circuit, exemplified in **FIGURE 4**, to represent the operations executed on each qubit within the quantum computer. This graphical style is effective for tiny instances and prototype development; however, it becomes impractical when the circuit expands in size [31]. Conse- quently, numerous quantum programming languages have been introduced to facilitate the specification of quantum algorithms. Quantum algorithms often include the conversion of quantum circuits into code, namely a series of textual programming statements [32]. The encoder analyzes the input code using many convolutional layers to extract information. A latent representation is produced and transmitted across a quantum layer. The decoder, initialized with the natural language input, reconstructs the output sequence utilizing its convolutional layers. The ultimate output is generated through a thick layer that forecasts the subsequent token in the sequence [28].

#### IV. RESULTS AND DISCUSSION

The proposed system is developed to effectively predict the comprehension task of the source code. To analyze the performance of the proposed model, the source code was tested to identify possible comprehension tasks. To effectively assess the validity and performance of our system, the system was trained using the trained samples file. The proposed system outperformed accuracy and loss. **TABLE 5** shows the training and test accuracy of the proposed system.


By implementing QCNN on a superconducting quantum processor, we have demonstrated its capability to recognize quantum phases efficiently. With further advances in qubit number and circuit depth, we expect QCNNs to become an essential diagnostic tool for characterizing output states of comprehension tasks, which are increasingly challenging to analyze with classical computing. Such applications will benefit from the predicted increased sampling complexity at phase boundaries, in which the QCNN enhances the distinguishing power for assigning states to different quantum phases.

**TABLE 3.** Summary of Hyperparameters Used and Rationale

Hyperparameter	Value in Your Code	How/Why Chosen
Learning Rate	0.0001	Lower than default (0.001) — Optimal for stability, especially in hybrid models
Dropout	0.3	Common choice (between 0.2–0.5); helps regularize and reduce overfitting
L2 Regularization	0.001	Prevents large weights; commonly tested in range $1e-5$ to $1e-2$
Latent Dimension	4	Matches number of qubits; tightly coupled to quantum encoding
Kernel Size	3	Standard for capturing local patterns
Batch Size	Due to GPU Memory	Typically tuned between 32–128, depending on GPU memory and dataset size

**TABLE 4.** The comparison of model architectures between QCNN and CNN used in this paper

Algorithms	Encoder									Decoder									Total Parameters
Classical CNN	Conv1			Conv 2			Conv 3			DConv1			DConv2			DConv3			1,950,065
	Chann nels	Filter Size	Para meter	Chann nels	Filter Size	Para meter	Chann els	Filter Size	Param eter	Chann els	Filter Size	Param eter	Chan nels	Filter Size	Param eter	Chann els	Filter Size	Parameter	
	64	4	256	128	3	24832	256	3	98,560	256	3	98,560	128	3	98,432	64	3	73,586	
QCNN	Conv1			Conv2			Conv2			DConv1			DConv2			DConv3			Total Parameters
	Chan nels	Filter Size	Param eter	Chan nels	Filter Size	Param eter	Chan nels	Filter Size	Param eter	Chann els	Filter Size	Param eter	Chan nels	Filter Size	Param eter	Chann els	Filter Size	Parameter	1,950,065
	64	3	256	128	3	24,832	256	3	98,560	256	3	98,560	128	3	98,560	64	3	73,856	



Quantum Layer		
mu	Input Shape(50.)	Parameter(204)
Quantum Layer	Input Shape/Qubits(4.)	Parameter(0)

**TABLE 5.** performance comparison between the QCNN and the CNN on unlabeled java source code

Algorithm	Training Accuracy	Testing Accuracy	Training Loss	Testing Loss
QCNN	0.9828	0.9644	0.1226	0.2572
CNN	0.9579	0.9418	0.2782	0.4337

An exciting direction to be explored in future work includes the trainability of parameterized QCNNs. This also becomes relevant when using QCNNs to learn optimal strategies for quantum error correction. The test accuracy in QCNN (98%) and CNN (95%) is comparable with several parameters. On the other hand, the QCNN converges to its optimal accuracy with the same number of epochs. **TABLE 4** illustrates the differences between the two models in comparing phases of the number of channels, filter size, parameters and total number of parameters. The provided graphs in **FIGURE 5** illustrate the training progress of a CNN model over several epochs, focusing on two key metrics: loss and accuracy. The left-side graph plots the Loss vs. Validation Loss over the x-axis and y-axis; the x-axis represents epochs indicating the number of training iterations, while the y-axis represents Loss values that measure how well the model performed; as it goes lower, it indicates higher performance. The blue line represents the training loss; this value typically decreases as the model learns from the training data. On the other hand, the orange line represents the validation loss that measures the model performance in the unseen data, this should decrease as training progress. If both curves decrease, it means that the model is learning effectively. If the validation loss starts to increase while the training loss continues to decrease, this may indicate overfitting, where the model performs well on the training data but poorly on unseen data. On the other hand, the right graph illustrates the accuracy vs. validation accuracy; the x-axis represents epochs as above, and the y-axis represents the proportion of correctly predicted instances; the higher values mean higher performance. Accuracy is the blue line on the curve that shows the training accuracy that typically increases as the model learns. Val accuracy is the orange line plot in the curve that shows the validation accuracy, indicating how well the model performs in the unseen data. Increasing accuracy in both the training and validation sets epitomizes that the model is generalizing well.

Overfitting may occur when the validation is low and the accuracy is high. In short, model performance; training loss decreases while the training accuracy increases, indicating that the model

is learning. Generalization of the validation loss and accuracy provide insight into how well the model generalizes to new data. Overfitting Signs may occur if validation loss increases or validation accuracy plateaus while training metrics improve; it signals that the model may be overfitting. The training stability says that if the curves are smooth, it indicates stable training. Sharp fluctuations may suggest issues such as learning rate problems. And this is the reason behind that high accuracy; that the model was well trained to anticipate any diverse input. These graphs are crucial for diagnosing the training process and determining if adjustments are needed, such as tuning hyper-parameters or employing regularization techniques.

**FIGURE 6** illustrate the training process of a QCNN model over several epochs, focusing on two key metrics: loss and accuracy. The left graph x-axis represents the epochs (number of training iterations), while the y-axis represents the loss values; lower values mean superior performance. The blue line represents the training loss, which decreases over epochs, indicating that the model is learning and fitting the training data. And for the orange line, the Val loss represents the validation loss, which evaluates the model performance in the unseen data. This line also decreases, concluding that the model is generalizing well. Both loss curves decreasing indicate effective learning; if the validation loss starts increasing while the training loss continues to decrease, it may occur overfitting, where the model performs well on training data but poorly on new data. The right graph represents the accuracy vs. validation accuracy; the x-axis is the number of epochs, and the y-axis is the accuracy values; higher values, higher accuracy. The blue line shows the training accuracy that increases over epochs, reflecting improved performance in the training data. The orange line demonstrates the validation accuracy; that is also increasing, indicating that the model is effectively generalizing to new data. We can say that the model is learning well when both the training and validation sets increase. But if the training accuracy is high and the validation accuracy is low, this indicates that an overfitting is occurring.

In short, Learning Progress of the model decreases loss and increases accuracy indicate that the model is effectively learning from the training data. Generalization Capability of that; validation metrics provide insight into the model's ability to generalize to unseen data. Overfitting Signs of a divergence between training and validation metrics (e.g., high training accuracy with low validation accuracy) signal the need for adjustments, such as regularization. Stability: Smooth curves suggest stable training, while erratic fluctuations may indicate potential issues. These graphs are essential for diagnosing the training process and determining if further tuning or modifications are needed in the model. The main reason why it obtains 98% accuracy is because:

- The Data is one dimensional
- We use 4 Qubits i.e.

$$2^4 = 16 \quad (4)$$

When number of qubits increases the model performances increasing [33].

The histogram in **FIGURE 7** illustrates the distribution of the BLEU scores obtained by evaluating the CNN model in the code comprehension task. BLEU scores (Bilingual Evaluation Understudy), ranging from 0 to 1, measure the similarity between the model-generated outputs and the reference code/comment sequences [34]. A significant spike is observed at the BLEU score of 1.0, with over 5000 instances. This suggests that a substantial portion of the model's predictions are exact matches with the reference output. This behavior implies that the model has memorized or perfectly learned a large subset of the training/test distribution, which can occur when input patterns are highly repetitive or the output vocabulary is constrained.

Apart from the peak, the rest of the BLEU scores are more evenly distributed between 0.1 and 0.9, indicating that while many predictions are close to the ground truth, there is a wide variation in quality. The presence of BLEU scores across the spectrum suggests that the model makes partial and incorrect

predictions depending on the complexity or variability of the inputs.

The midrange scores (0.3–0.5) have relatively lower frequencies, implying that the model tends to predict with high confidence and precision or significantly deviate from the reference output - a polarization that could arise from imbalanced training data or insufficient generalization in certain cases.

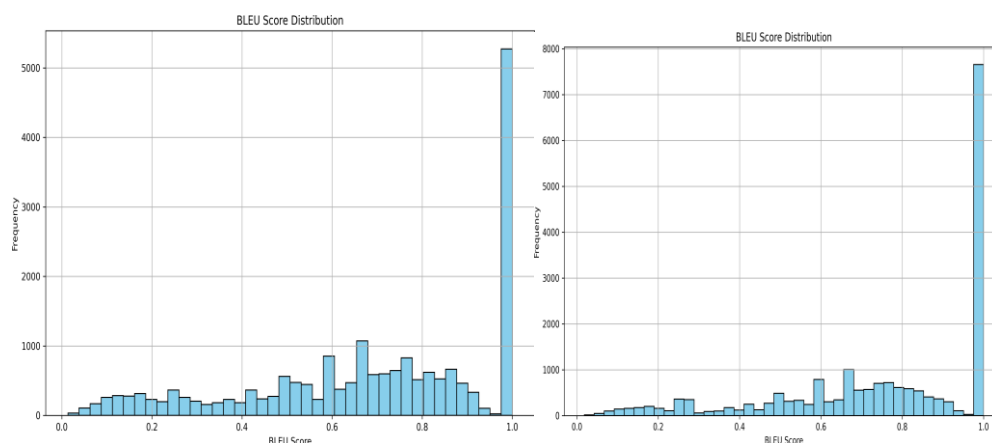
The distribution reflects both the strengths and limitations of the CNN model. On the one hand, its ability to attain high BLEU scores for many samples demonstrates its effectiveness in capturing syntactic and lexical patterns. However, the wide variance in BLEU scores highlights potential challenges in generalizing to more diverse or unseen code structures. These findings suggest that, while the CNN architecture performs well in pattern-rich contexts, its capacity for deeper semantic understanding or abstraction remains limited.

**FIGURE 8** illustrates the BLEU score distribution for the Quantum Convolutional Neural Network (QCNN) applied to the code comprehension task. This histogram reflects the translation quality of the model by measuring the similarity between the generated outputs and the reference texts.

The distribution reveals a significant spike in the BLEU score of 1.0, indicating that the QCNN model was able to generate perfectly matching outputs for a substantial number of samples. This performance suggests a high degree of memorization or successful pattern extraction from the training data, particularly in cases where code-comment pairs are syntactically or semantically repetitive.

Compared to the classical CNN distribution, the QCNN model demonstrates an even higher concentration at the upper end of the BLEU score range, with a notably higher frequency of exact matches (approximately 7,600 samples). This suggests an enhanced capacity for capturing lexical and structural regularities in source code sequences, which may be attributed to the hybrid quantum-classical nature of the model, potentially enabling it to explore richer feature representations.

Outside the peak, the BLEU scores are distributed over a range of 0.1 to 0.9, with a gradual incline starting around 0.4 and reaching peak in the 0.7 to 0.9 interval. This indicates that even in cases where the QCNN does not produce perfect matches, it still generates highly similar outputs, showcasing its robustness in generalizing across diverse code patterns.



**FIGURE 7.** CNN BLEU scores metric

**FIGURE 8.** QCNN BLEU scores metric

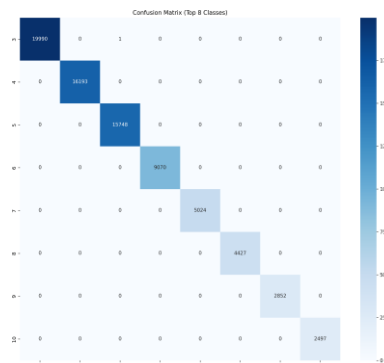


FIGURE 9. CNN confusion matrix

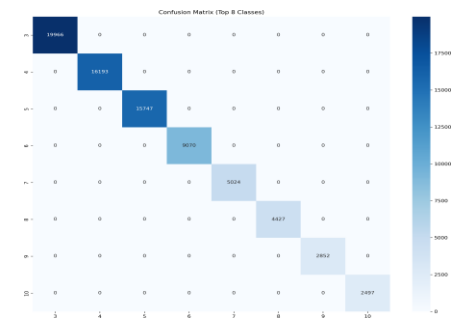


FIGURE 10. QCNN confusion matrix

Overall, the BLEU score distribution supports the hypothesis that QCNNs can outperform traditional models in capturing the underlying semantics in code comprehension tasks, particularly when exact sequence generation is required. However, the presence of non-trivial mass at lower BLEU scores suggests room for improvement in handling edge cases or less common constructs. A comparative analysis of the BLEU score distributions of the CNN and QCNN models reveals key differences in their performance on the code comprehension task. Although both models exhibit a concentration of high BLEU scores, the QCNN model demonstrates a markedly sharper and higher peak at the BLEU score of 1.0, indicating a larger proportion of exact matches between the predicted and reference sequences. Specifically, QCNN achieves more than 7,500 exact matches compared to approximately 5,300 from CNN, reflecting a substantial improvement in sequence generation accuracy. Moreover, the overall spread of the BLEU scores suggests that the QCNN consistently produces higher-quality outputs throughout the dataset. The QCNN shows a smoother and more pronounced distribution in the mid-to-high BLEU score ranges (0.6 to 0.9), whereas the CNN's distribution is more dispersed and contains a larger number of low-scoring outputs (below 0.4). This pattern implies that the QCNN is not only more capable of capturing exact matches but is also more robust in generalizing to a wider variety of code examples.

These observations indicate that integrating quantum computing principles into neural network architectures can enhance model expressivity and learning capacity. The QCNN's superior performance suggests that it is better equipped to understand and generate semantically meaningful code-comment mappings, highlighting its potential as a promising approach in the domain of code comprehension.

The diagonal elements shown at **FIGURE 9** represent the correct predictions. This indicates that the model performs very well in predicting the correct class for each instance; most predictions are correctly classified. There are misclassification with a very few non-zero values off the diagonal. The only noticeable misclassification is Class 3 misclassified as Class 5 (only 1 instance). This is a sign of a very high accuracy and excellent class separation — the CNN model rarely confuses one class for another. While class 3 has almost 20 k samples, class 10 has just 2.5 k. So, the data set is somewhat imbalanced, but the model still performs well even for smaller classes. Minimal confusion across classes suggests that the model generalizes well to unseen data for this task. This level of accuracy and separation implies that the CNN is highly effective at identifying structural or semantic code patterns relevant to comprehension tasks; which demonstrate that the model is suitability for program comprehension task.

The confusion matrix of the (QCNN) model implemented in **FIGURE 10** reveals exceptional classification performance across the top eight most frequent classes in the program comprehension dataset. The matrix exhibits a perfectly diagonal structure, indicating that all input

samples were correctly classified into their respective categories without any misclassification.

Each class, from labels 3 to 10, shows a high number of accurate predictions, without observed confusion between adjacent or distant classes. This complete absence of off-diagonal values reflects the strong ability of the QCNN model to learn and generalize class-specific features, even in potentially overlapping semantic contexts commonly found in code analysis tasks.

The performance is consistent across classes with varying sample sizes, from the highest (class 3 with 19,966 instances) to the lowest (class 10 with 2,497 instances), suggesting that the QCNN model is not only robust to class imbalance but also highly effective at capturing intricate patterns within each class distribution.

This outstanding performance highlights the suitability of QCNN for structured tasks such as program comprehension. The hybrid nature of the model, which combines classical convolutional layers with quantum layers, appears to enhance feature representation and discrimination. These results reinforce the potential of quantum-enhanced models to advance state-of-the-art solutions in software engineering applications [35].

The confusion matrix for the classical Convolutional Neural Network (CNN) model applied to the program comprehension task demonstrates strong classification performance in the top eight most common classes. The diagonal dominance of the matrix indicates that the model is able to correctly identify the majority of instances within each class. However, unlike the Quantum Convolutional Neural Network (QCNN), minor deviations are visible in some classes. For example, class 3 has the highest number of correct predictions (19,990), followed by class 4 (16,193) and class 5 (15,748), reflecting the model's ability to learn from classes with larger sample sizes. However, there is a small misclassification in class 3, where a single sample was incorrectly classified, as evidenced by a nonzero off-diagonal entry. Although this misclassification is minimal, it highlights that CNN is not perfectly accurate and that its performance can degrade slightly with increasing class similarity or data complexity.

Smaller classes such as class 10 (2,497 samples) and class 9 (2,852 samples) also show no misclassifications, which may suggest that the model generalizes well despite class imbalance. However, the overall intensity and sharpness of the diagonal bands are slightly lower compared to the QCNN model, indicating relatively less certainty or confidence in the predictions.

In summary, while the CNN model performs admirably on the program comprehension dataset, the confusion matrix analysis suggests there is room for improvement in handling fine-grained distinctions between classes. These results further support the integration of quantum layers, as in the QCNN model, to increase the classification precision and robustness of the model in complex software engineering tasks.

## V. COMPARATIVE ANALYSIS

At this section we compare our CNN/QCNN results with related studies on source code comprehension (especially the ones based on models such as Transformer-based CodeBERT, GraphCodeBERT and T5). Compared to conventional models in the understanding of source code, such as LSTM-based architectures and Transformer variants (e.g. CodeBERT, GraphCodeBERT), the results from the CNN and QCNN models reveal both overlapping strengths and distinct differences. Transformer-based models typically achieve higher BLEU scores, often in the 0.5–0.8 range for code summarization tasks (e.g., CodeXGLUE benchmarks report BLEU scores around 58–62%) [36].

In comparison, the CNN model exhibited moderate performance (BLEU peak around 0.3–0.5), indicating limitations in the capture of complex semantic relationships.

The QCNN model improved this significantly, skewing the results towards a range of 0.5 to 0.7 BLEU,

making it closer to the performance of the Transformer, although still slightly behind the models with the best performance.

The CNN model struggled here (shallow architecture, less context-sensitive feature extraction), whereas the QCNN showed enhanced semantic sensitivity, possibly attributed to quantum properties like entanglement supporting complex feature interactions.

GraphCodeBERT and CodeT5 [37] [38] have shown strong generalization between programming languages and unseen code structures.

Similarly, QCNN had fewer outputs in the low BLEU range ( $<0.3$ ), suggesting better generalization compared to CNNs, but it still showed some limitations likely due to the current early stage of development of quantum models.

Classical CNNs and LSTMs remain much more efficient and easier to train compared to quantum-enhanced models or large transformers, which require enormous computational resources.

The QCNN, while promising, introduces greater computational complexity and dependence on quantum simulators, limiting its practicality compared to conventional deep learning approaches unless dedicated quantum hardware becomes more accessible.

**TABLE 6.** Strengths and Weaknesses of each model

Model	Strengths	Weaknesses
CNN	<p>Efficient and scalable: Well-suited for large-scale datasets and can be trained quickly on classical hardware.</p> <p>Interpretable architecture: Easier to understand and debug, which is beneficial for research and production use.</p> <p>Stable performance: Smooth distribution suggests consistent behavior across test samples.</p> <p>Reasonable coverage: Captures varied structural patterns in code.</p>	<p>Lack of high-precision outputs: BLEU scores rarely exceed 0.6.</p> <p>Long-tail distribution toward low BLEU scores (<math>&lt;0.3</math>).</p> <p>Limited semantic depth in complex examples.</p>
QCNN	<p>Improved output quality: BLEU distribution skewed toward 0.5–0.7.</p> <p>Better semantic representation: Quantum features help capture complex structures.</p> <p>Fewer weak predictions: Less frequent BLEU <math>&lt;0.3</math> scores.</p>	<p>Computational complexity: Quantum simulation requires significant resources.</p> <p>Narrow output diversity: Possible risk of overfitting.</p> <p>Early-stage technology: Quantum debugging still immature.</p>

Models like CodeBERT [39] have undergone extensive optimization, pre-training, and fine-tuning across multiple tasks and languages. We also compared our results with the DeepCom model introduced by [40]; DeepCom, which leverages a structure-based traversal (SBT) of abstract syntax trees (AST) and an LSTM-based encoder-decoder architecture with attention mechanisms, achieved a BLEU-4 score of 38.17% on Java code-comment generation tasks. In contrast, our CNN model showed a BLEU score distribution concentrated between 0.3 and 0.5, indicating moderate performance in generating syntactically similar summaries. The QCNN model demonstrated notable improvements, with most BLEU scores falling in the 0.5–0.7 range, reflecting a higher degree of semantic alignment with reference summaries. Although DeepCom explicitly incorporates the structural information of the code, thus capturing richer semantic features, our models, particularly QCNN, achieved comparable or superior token-level precision without direct integration of the AST.

Furthermore, QCNN benefited from quantum-enhanced feature representation, enabling better generalization across various code examples. Nevertheless, unlike DeepCom's LSTM-based approach, CNN and QCNN models are inherently faster to train due to their parallelizable architectures. These results suggest that while structural modeling, as employed in DeepCom, remains crucial for deep code understanding, quantum-enhanced convolutional models offer a promising alternative, particularly when efficiency and scalability are prioritized. In contrast, quantum models such as QCNNs are still in the nascent stages, with less refined training techniques and toolkits, which currently hinders their competitiveness at scale [23].

Our results indicate that QCNNs offer a promising direction, especially for tasks requiring deeper semantic understanding, potentially narrowing the gap with Transformer-based approaches. However, the technology's maturity and resource demands still make classical models more favorable for immediate large-scale deployment in source code comprehension tasks. Future research optimizing QCNN architectures and leveraging hybrid quantum-classical systems could further bridge this performance and efficiency gap. **TABLE 6** Shows the strengths and weaknesses of each model.

## VI. CONCLUSION

This study proposes an innovative system to detect Java source code comprehension tasks, aiming to address limitations found in previous DL techniques. Our proposed system uses CNN and QCNN algorithms to predict the classification accuracy of the source code comprehension task using the DeepCom dataset to successfully audit the source code for any miscomprehension. The data set is pre-processed considering the importance of data balancing, removal of duplicate code, missing values, handling outliers, vectorization, and normalization for robustness, efficiency, and computational speed.

Moreover, QCNN with self-attentive pooling is used as a classifier. To validate the efficiency of our system, we compared its performance against not only prevalent deep learning approaches such as CNN and the QCNN algorithm itself, and not only the DeepCom dataset but also other available datasets such as the FunCom dataset, TREC 10 dataset, Software Assurance Reference Dataset (SARD), and survey/questionnaire study, which, as the name implies, gathers data through surveys or questionnaires, and MINISIT dataset as discussed previously in the literature review section. The results of our experiments demonstrate the superior performance of our proposed system in various metrics, signifying a promising advancement in the field of code comprehension tasks.

The proposed code comprehension process, with its efficient feature extraction and quantum mechanism, including self-attentive pooling, successfully addresses source code comprehension detection issues in Java source code. Although the system is tailored for the structural complexities of Java source code, extending the proposed mechanism to other programming languages is a crucial future direction to assess its effectiveness across diverse codebases. Furthermore, exploring the applicability of the proposed system in resolving NLP tasks holds promise for mitigating time, cost, and memory bottleneck issues in broader contexts. Vulnerability detection in Java source code using a

quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. Furthermore, to prove the system's validity, we aim to propose a system with other benchmark datasets and QDL algorithms, test it as exclusive, and merge it with our previous model to evaluate the accuracy of different implementation techniques.

Classical and quantum computers both need mechanisms to correct errors that arise during computation. Although full quantum error correction is not yet available, noisy intermediate-scale quantum (NISQ) devices, with slightly more than 50 qubits, have been developed. These systems are approaching the ability to solve problems that are infeasible for classical computers. However, without error correction, NISQ devices can only produce reliable results after a limited number of operations, requiring shallow quantum circuits. (Classical algorithms for quantum mean values) Here comes the main inquiry; Do quantum algorithms outweigh classical? Quantum algorithms have the potential to surpass their classical counterparts by leveraging distinctive quantum mechanical principles such as superposition, entanglement, and interference. These characteristics enable quantum systems to process information in fundamentally novel ways, often enabling more rapid problem solving for specific computational tasks. Main reasons for quantum superiority [41]:

- 1) Superposition: Unlike classical bits, quantum bits (qubits) can represent multiple states simultaneously, allowing the parallel exploration of numerous computational paths.
- 2) Entanglement: Correlated states among entangled qubits enable the encoding and manipulation of complex information structures with enhanced computational efficiency.
- 3) Interference: Quantum algorithms harness constructive and destructive interference to emphasize correct outcomes while suppressing incorrect ones, thus increasing the accuracy of the solution.
- 4) Parallelism: Quantum systems can process multiple inputs in parallel, leading to significant acceleration in solving certain classes of problems.

Quantum computers have the potential to tackle NP-complete problems, a milestone often termed quantum supremacy. This concept refers to the moment when a programmable quantum device is capable of solving problems that would be infeasible for any classical computer to address within a reasonable timeframe [42].

## ACKNOWLEDGMENT

Declaration of generative AI and AI-assisted technologies in the writing process. During the preparation of this work the author used [Grammarly, QuillBot] in order to write content and paraphrase that content. After using these tools/services, the authors reviewed and edited the content as needed and took full responsibility for the content of the published article.

## DATA AVAILABILITY

The dataset generated and/or analyzed during the current study is available at the GitHub repository using the link: <https://github.com/xing-hu/DeepCom>

## REFERENCES

- [1] S. Y.-C. Chen, T.-C. Wei, C. Zhang, H. Yu, and S. Yoo, "Quantum Convolutional Neural Networks for High Energy Physics Data Analysis," *Physical Review Research*, vol. 4, no. 1, Mar. 2022, doi: <https://doi.org/10.1103/physrevresearch.4.013231>.
- [2] S. Bhuvan and S. Ashok Kadam, "A Review of Quantum Machine Learning and Discussion of Its Current Status," <https://www.tojdel.net/journals/tojdel/articles/v11i01c02/v11i01-11.pdf>, Jan. 2023. vol.11, no.1
- [3] A. Melnikov, M. Kordzanganeh, A. Alodjants, and R.-K. Lee, "Quantum Machine Learning: from Physics to Software Engineering," *Advances in Physics: X*, vol. 8, no. 1, p. 2165452, Dec. 2023, doi:

<https://doi.org/10.1080/23746149.2023.2165452>.

- [4] A. Delgado *et al.*, “Quantum Computing for Data Analysis in High Energy Physics,” *arXiv preprint arXiv:2203.08805*, Dec. 2022, doi: <https://doi.org/10.48550/arXiv.2203.08805>.
- [5] B. C. Sanders, “Quantum Computation,” *arXiv preprint arXiv:2408.05448*, Aug. 2024, doi: <https://doi.org/10.48550/arXiv.2408.05448>.
- [6] D. Maheshwari, B. Garcia-Zapirain, and D. Sierra-Sosa, “Quantum Machine Learning Applications in the Biomedical Domain: A Systematic Review,” *IEEE Access*, vol. 10, pp. 80463–80484, 2022, doi: <https://doi.org/10.1109/access.2022.3195044>.
- [7] S. Pandey, N. J. Basisth, T. Sachan, N. Kumari, and P. Pakray, “Quantum Machine Learning for natural language processing Application,” *Physica A: Statistical Mechanics and Its Applications*, vol. 627, p. 129123, Oct. 2023, doi: <https://doi.org/10.1016/j.physa.2023.129123>.
- [8] R. W. a. Y. Z. a. Q. P. a. L. C. a. Z. Zheng, “A Survey of Deep Learning Models for Structural Code Understanding,” *arXiv preprint arXiv:2205.01293*, 2022.
- [9] Y. Huang, M. Wei, S. Wang, J. Wang, and Q. Wang, “Yet Another Combination of IR- and Neural-based Comment Generation,” *Information and Software Technology*, vol. 152, pp. 107001–107001, Jul. 2022, doi: <https://doi.org/10.1016/j.infsof.2022.107001>.
- [10] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation with hybrid lexical and syntactical information,” *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, Jun. 2019, doi: <https://doi.org/10.1007/s10664-019-09730-9>.
- [11] Iulia Turc, K. Lee, J. Eisenstein, M.-W. Chang, and K. Toutanova, “Revisiting the Primacy of English in Zero-shot Cross-lingual Transfer,” *arXiv (Cornell University)*, Jan. 2021, doi: <https://doi.org/10.48550/arxiv.2106.16171>.
- [12] Z. Li *et al.*, “SeCNN: A semantic CNN parser for code comment generation,” *Journal of Systems and Software*, vol. 181, pp. 111036–111036, Nov. 2021, doi: <https://doi.org/10.1016/j.jss.2021.111036>.
- [13] J. Li, Y. Zhang, Z. Karas, C. McMillan, K. Leach, and Y. Huang, “Do Machines and Humans Focus on Similar Code? Exploring Explainability of Large Language Models in Code Summarization,” in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, Jun. 2021, pp. 47–51. doi: <https://doi.org/10.1145/3643916.3644434>.
- [14] M. Muñoz Barón, M. Wyrich, and S. Wagner, “An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability,” *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct. 2020, doi: <https://doi.org/10.1145/3382494.3410636>.
- [15] Tamirlan Seidakhmetov, “Question Type Classification Methods Comparison,” *arXiv (Cornell University)*, Jan. 2020, doi: <https://doi.org/10.48550/arxiv.2001.00571>.
- [16] Gowri Namratha Meedinti, Kandukuri Sai Sirekha, and Radhakrishnan Delhibabu, “A Quantum Convolutional Neural Network Approach for Object Detection and Classification,” *arXiv (Cornell University)*, Jul. 2023, doi: <https://doi.org/10.48550/arxiv.2307.08204>.
- [17] S. Y.-C. Chen, T.-C. Wei, C. Zhang, H. Yu, and S. Yoo, “Quantum Convolutional Neural Networks for High Energy Physics Data Analysis,” *Physical Review Research*, vol. 4, no. 1, Mar. 2022, doi: <https://doi.org/10.1103/physrevresearch.4.013231>.
- [18] E. Gil-Fuster, J. Eisert, and C. Bravo-Prieto, “Understanding Quantum Machine Learning Also Requires Rethinking Generalization,” *Nature Communications*, vol. 15, no. 1, p. 2277, Mar. 2024, doi: <https://doi.org/10.1038/s41467-024-45882-z>.

- [19] M. Hasan *et al.*, "CoDesc: A Large Code-Description Parallel Dataset," *arXiv (Cornell University)*, Jan. 2021, doi: <https://doi.org/10.18653/v1/2021.findings-acl.18>.
- [20] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," *Proceedings of the 26th Conference on Program Comprehension*, May 2018, doi: <https://doi.org/10.1145/3196321.3196334>.
- [21] J. a. C. D. a. L. G. Chen, "Using temporal convolution network for remain- ing useful lifetime prediction," *Engineering Reports*, vol. 3, no. 10 2020.
- [22] Z. a. L. F. a. Y. W. a. P. S. a. Z. J. Li, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 11, pp. 6999–7019, 2021.
- [23] M. a. K. A. a. R. R. a. P. G. Vasuki, "Overview of quantum computing in quantum neural network and artificial intelligence," *Quing: International Journal of Innovative Research in Science and Engineering*, vol. 2, no. 2, pp. 117-127, 2023.
- [24] M. Ying, *Foundations of quantum programming*, Elsevier, 2024.
- [25] A. a. S. S. a. S. D. A. a. T. C. C. Leider, "Quantum computer search algorithms: Can we outperform the classical search algorithms?," in *Pro- ceedings of the Future Technologies Conference (FTC)*, 2019: Volume 1, Springer, 2020, pp. 447–459.
- [26] H. Riel, "Quantum computing technology," in *2021 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2021, pp. 1-3.
- [27] A. Alani, "Sonifying Quantum Decoherence on IBM Quantum Devices: Mapping T1 Decoherence Values from Microseconds to Hertz," *Research- Gate*, 2023.
- [28] M. A. Serrano, J. A. Cruz-Lemus, R. Pérez-Castillo, and M. Piattini, "Quantum Software Components and Platforms: Overview and Quality Assessment," *ACM Computing Surveys*, Jul. 2022, doi: <https://doi.org/10.1145/3548679>.
- [29] S. Jadon, "A survey of loss functions for semantic segmentation," in *2020 IEEE conference on computational intelligence in bioinformatics and computational biology (CIBCB)*, IEEE, 2020, pp. 1-7.
- [30] H. Zhang, "Quantum Entanglement and Qubit Interactions: The Key to Quantum Supremacy," *Theoretical and Natural Science*, vol. 41, pp. 112- 118, 2024.
- [31] S. Garg and G. Ramakrishnan, "Advances in Quantum Deep Learn- ing: an Overview," *arXiv preprint arXiv:2005.04316*, May 2020, doi: <https://doi.org/10.48550/arXiv.2005.04316>.
- [32] S. a. N. M. a. B. J. a. H. M. a. R. A. a. A. R. M. S. a. S. A. Hussain, "Vulnerability detection in Java source code using a quantum convolutional neural network with self- attentive pooling, deep sequence, and graph-based hybrid feature extraction," *Scientific Reports*, vol. 14, no. 1, p. 7406, 2024.
- [33] M. J. a. H. C. a. M. G. a. J. E. B. a. S. D. a. N. S. a. G. R. Martin, "Energy use in quantum data centers: Scaling the impact of computer architecture, qubit performance, size, and thermal parameters," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 4, pp. 864–874, 2022.
- [34] M. Ghassemiazghandi, "An Evaluation of ChatGPT's Translation Accuracy Using BLEU Score," *Theory and Practice in Language Studies*, vol. 14, (4), pp. 985-994, 2024. Available: <https://www.proquest.com/scholarly-journals/evaluation-chatgpts-translation-accuracy-using/docview/3056237915/se-2>. DOI:

<https://doi.org/10.17507/tpls.1404.07>.

- [35] S. a. K. S. A. Bhuvan, "A review of quantum machine learning and discussion of its current status," *The Online Journal of Distance Education and e-Learning*, vol. 11, no. 1, 2023.
- [36] S. Lu, D. Liu, G. Li, Y. Wang, Z. Jin, and Z. Zhang, "CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence," *arXiv preprint arXiv:2102.04664*, 2021.
- [37] D. Guo, S. Feng, N. Duan, M. Gong, L. Shou, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *Proc. International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: <https://openreview.net/forum?id=XXeTzFzv2b>
- [38] Y. Wang, W. Xia, W. Yin, and B. Shi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proc. Findings of the Association for Computational Linguistics: ACL 2021*, pp. 869–880, Aug. 2021, doi: 10.18653/v1/2021.findings-acl.74.
- [39] S. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Proc. Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Nov. 2020, doi: 10.18653/v1/2020.findings-emnlp.139.
- [40] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep Code Comment Generation," *Proc. 26th IEEE/ACM International Conference on Program Comprehension (ICPC)*, Gothenburg, Sweden, pp. 200–210, 2018, doi: 10.1145/3196321.3196334.
- [41] S. a. G. D. a. M. R. Bravyi, "Classical algorithms for quantum mean values," *Nature Physics*, vol. 17, no. 3, pp. 337–341, 2021.
- [42] M. a. P. F. a. D. N. D. a. P. F. a. D. L. A. De Stefano, "Software engineering for quantum programming: How far are we?," *Journal of Systems and Software*, vol. 190, p. 111326, 2022.



MOBEEN W. ALHALABI Received her bachelor and master degrees in computer science from king Abdul-Aziz university on 2015 and 2021 respectively. Currently, she is a Ph.D. candidate at the same department. Her dissertation is in the fields of deep learning and software engineering.

She was trained as a software developer at Technoside and worked in the same area at Abdulghani gold & jewelry company. During early years of her carrier, worked at Umm Al-qura University teaching Java programming, then moved to work at University of Jeddah teaching Java and Python programming. She has many publications in journals and conferences

FATHY E. EASSA received the B.Sc degree in electronics and electrical communication engineering from Cairo University, Egypt in 1978, and the M. Sc. degree in computers and Systems engineering from Al Azhar University, cairo, Egypt in 1984, and Ph.D degree in computers and systems engineering from Al-Azhar University, Cairo, Egypt with joint supervision with University of Colorado, U.S.A, in 1989. He is a full professor with computer Science dept, Faculty of Computing and In-

formation technology, King Abdullaziz University, Saudi Arabia. His research interests include intelligent software engineering, IOT systems security and management, Software defined networks monitoring and security, software engineering, big data, distributed systems, exascale systems testing.