Advanced Deep Neural Network For Enhancing And Evaluating Programming Performance

Yue Wang*

Faculty of Teacher Education, Qilu Normal University, 2 Wenbo Road, Jinan 250200, Shandong, China

Corresponding author. E-mail: wangyue6775@163.com

Received: Mar. 07, 2026; Accepted: Mar. 27, 2026

With the rapid development of intelligent software engineering and computer science education, automatic programming quality enhancement and quantitative programming performance evaluation have become increasingly critical research directions. Traditional evaluation approaches mainly rely on manual scoring, static code checking tools, and limited test case execution, which are inefficient, subjective, and incapable of capturing deep semantic information and long-range logical dependencies in source code. Meanwhile, existing code optimization methods focus on single tasks such as bug fixing or code summarization, lacking a unified framework that supports both code enhancement and comprehensive performance assessment. To address these limitations, this paper proposes a novel end-to-end deep learning framework named CodeProNet for jointly enhancing programming quality and evaluating programming performance. The model integrates multi-modal feature extraction, semantic-aware graph representation, multi-scale Transformer encoding, and contrastive-learning-based performance prediction. Specifically, we design a semantic-structure fused code representation that combines lexical sequence information, abstract syntax tree (AST) structure, and data-flow graph (DFG) semantics to fully encode intrinsic characteristics of source code. A multi-scale Transformer encoder is introduced to capture both local syntactic patterns and global logical dependencies. Furthermore, a dual-task learning mechanism is constructed to simultaneously optimize code enhancement and performance evaluation. Extensive experiments are conducted on three representative datasets: CodeSearchNet, HumanEval, and a self-built enterprise-level annotated programming dataset (Enterprise Programming Dataset (EPD)). Quantitative results demonstrate that CodeProNet achieves 92.3% accuracy in programming performance grading, 13.7% code error rate, and 85.7% Pass@1 in code functional correctness, significantly outperforming baseline models including CodeBERT, GraphCodeBERT, and CodeT5. Ablation studies verify the effectiveness of each core component. This work provides a unified, scalable, and interpretable solution for intelligent programming education, automated code review, and developer capability evaluation.

Keywords: Deep Neural Networks; Programming Performance; Source Code Representation; Code Enhancement; Contrastive Learning; Intelligent Software Engineering

© The Author('s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are cited.

http://dx.doi.org/10.6180/jase.202609_32.003

1. Introduction

In the era of artificial intelligence and big data, programming has become a fundamental skill for students, engineers, and researchers in computer science, software engineering, data science, and interdisciplinary fields. The scale and complexity of software systems are growing exponentially, which places higher demands on programming efficiency, code quality, algorithm logic, and program robustness. Therefore, how to automatically enhance pro-

gramming quality and objectively evaluate programming performance has become an urgent problem in both academic research and industrial applications [1–3].

Early source code modeling methods were based on traditional machine learning and shallow neural networks. With the rise of deep learning, convolutional neural network (CNN), long-short term memory (LSTM), Gated Recurrent Unit (GRU), and other structures were gradually applied to capture sequential and local features of code. However, these models have limitations in modeling long-range dependencies and structural information. Transformer-based pre-trained models have become mainstream in code intelligence. CodeBERT [4] uses a hybrid pre-training strategy of programming language and natural language to learn general code representation. GraphCodeBERT [5] further integrates data-flow graphs to enhance semantic understanding. CodeT5 [6] adopts a unified encoder-decoder framework for multi-task programming learning. Although these models improve code representation ability, they rarely consider the joint optimization of code enhancement and performance evaluation.

Traditional programming training and evaluation systems suffer from several inherent drawbacks. First, manual evaluation is time-consuming, labor-intensive, and prone to subjective bias. Instructors or reviewers need to repeatedly read and judge code logic, which is inefficient for large-scale programming tasks. Second, traditional static analysis tools only detect syntax errors, code style issues, and simple semantic errors, but cannot understand deep program logic, algorithm efficiency, or code readability. Third, dynamic testing based on test cases can only judge whether the program passes execution, but cannot provide a comprehensive quantitative score for programming ability [7–9]. Fourth, most existing deep learning-based code models are designed for a single task, such as code generation, defect detection, or code retrieval, and rarely integrate code enhancement and performance evaluation into a unified framework.

Some studies attempt to use machine learning for automatic programming grading. However, most of them rely on handcrafted features or simple neural networks, which cannot capture deep semantic and logical features. With the development of pre-trained models, a few works attempt to apply code models to evaluation tasks, but they lack unified end-to-end frameworks and interpretable scoring mechanisms.

Code enhancement covers code repair, code refactoring, code optimization, readability improvement, and bug fixing [10]. Sequence-to-sequence models and pre-trained models have been used to generate optimized code [11]. However, existing methods usually focus on functional correctness rather than comprehensive programming quality, and few works integrate enhancement with evaluation to form a closed-loop intelligent system.

To solve the above research gaps, this paper proposes CodeProNet (Code Performance Network), an advanced deep neural network for unified programming enhancement and evaluation. The main contributions are summarized as follows:

(1) We propose an end-to-end deep learning framework CodeProNet that supports both automatic code enhancement and quantitative programming performance evaluation in a unified model.

(2) We design a semantic-structure fused code representation that combines lexical sequence, abstract syntax tree, and data-flow graph to comprehensively encode code characteristics.

(3) We introduce a multi-scale Transformer encoder to capture multi-grained features and a contrastive-learning-based scoring module to improve the stability and interpretability of evaluation.

(4) We conduct extensive experiments on public datasets and a real-world dataset, achieving state-of-the-art performance compared with existing SOTA models. We also provide in-depth ablation studies and visualization analysis to verify the rationality and effectiveness of each module.

2. Materials and methods

This section describes the overall architecture and each core module of CodeProNet in
detail.

2.1. Overall Architecture

CodeProNet is a unified end-to-end dual-task deep neural network. The model consists of five modules: code preprocessing and tokenization, semantic-structure fused code representation, multi-scale transformer encoder, code enhancement decoder, contrastive-learning-based programming performance evaluator.

The framework takes raw source code as input and outputs two results simultaneously: enhanced and optimized code, quantitative programming performance score. Fig. 1 illustrates the complete pipeline of CodeProNet.

2.2. Code Preprocessing and Tokenization

First, source code is normalized through the following steps:

(1) Remove redundant spaces, comments, and blank lines;

(2) Unify variable naming format;

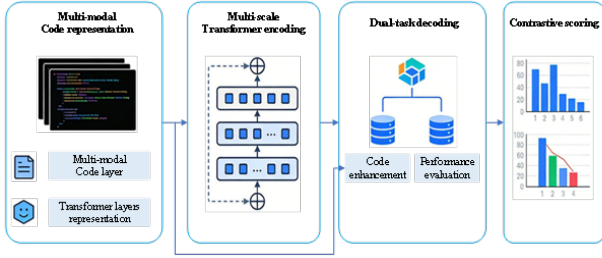

Fig. 1. Overall architecture of CodeProNet.

(3) Convert code into a standard token sequence;
(4) Parse AST structure and extract data-flow graph.

We use a byte-level BPE tokenizer [12, 13] adapted to programming languages to convert code into discrete token sequences. The tokenized sequence is mapped to embedding vectors as shown equation (1):

$$E = \text{Embedding}(T) \tag{1}$$

Where $T = [t_1, t_2, \ldots, t_n]$ represents the code token sequence, and $E \in \mathbb{R}^{n \times d}$ is the token embedding matrix.

2.3. Semantic-Structure Fused Code Representation

To fully capture lexical, syntactic, and semantic information, we fuse three types of code features:
(1) Sequential Lexical Feature H_{seq} from token embedding and position encoding;
(2) Structural Syntactic Feature H_{ast} from abstract syntax tree;
(3) Semantic Dependence Feature H_{flow} from data-flow graph.

The fused feature is computed as a weighted combination:

$$H_{\text{base}} = \alpha \cdot H_{\text{seq}} + \beta \cdot H_{\text{ast}} + \gamma \cdot H_{\text{flow}} \tag{2}$$

Where α, β, γ are learnable weights satisfying $\alpha + \beta + \gamma = 1$.

To enhance fusion effectiveness, we use a gated fusion mechanism:

$$g = \sigma\left(W_g \cdot \left[H_{\text{seq}} ; H_{\text{ast}} ; H_{\text{flow}}\right] + b_g\right) \tag{3}$$

$$H_{\text{fuse}} = g \odot H_{\text{seq}} + (1 - g) \odot \left(W_{\text{fusion}} \cdot \left[H_{\text{ast}} ; H_{\text{flow}}\right]\right) \tag{4}$$

Where g is the fusion gate. σ is the sigmoid function, and $\odot$ denotes element-wise multiplication.

2.4. Multi-Scale Transformer Encoder

We design a multi-scale Transformer encoder to capture local and global code features.

The multi-head self-attention mechanism is defined as:

$$\text{Head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{5}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{Head}_1, \ldots, \text{Head}_h\right) W^O \tag{6}$$

To enhance multi-scale perception, we use convolution-augmented attention:

$$H_{\text{local}} = \text{Conv}\,1d\left(H_{\text{fuse}}\right) \tag{7}$$

$$H_{\text{global}} = \text{MultiHead}\left(H_{\text{fuse}}, H_{\text{fuse}}, H_{\text{fuse}}\right) \tag{8}$$

$$H_{\text{enc}} = \text{LayerNorm}\left(H_{\text{local}} + H_{\text{global}}\right) \tag{9}$$

The final encoded feature H_{enc} contains both local syntactic patterns and global logical structures.

2.5. Code Enhancement Decoder

The code enhancement module adopts a pointer-augmented seq2seq decoder:

$$p_{\text{vocab}} = \text{Softmax}\left(W_o H_t + b_o\right) \tag{10}$$

$$p_{\text{gen}} = \sigma\left(w_h^T H_t + w_x^T x_t + w_c^T c_t + b_{\text{gen}}\right) \tag{11}$$

$$P_{\text{final}} = p_{\text{gen}}\, p_{\text{vocab}} + \left(1 - p_{\text{gen}}\right) p_{\text{pointer}} \tag{12}$$

This structure allows the model to both generate optimized code and copy necessary
tokens from the input, improving code correctness and readability.

2.6. Contrastive-Learning-Based Performance Evaluator

To improve evaluation stability and discrimination, we introduce contrastive learning.

We define that z_i is the representation of code i, z_i^+ is positive sample (high-quality code with similar logic). z_i^- is negative samples (low-quality or logically different code).

The contrastive loss is:

$$\mathcal{L}_{\text{cont}} = -\log \frac{\exp\left(\text{sim}\left(z_i, z_j^+\right)/\tau\right)}{\exp\left(\text{sim}\left(z_i, z_j^+\right)/\tau\right) + \sum_{j \neq i} \exp\left(\text{sim}\left(z_i, z_j\right)/\tau\right)} \tag{13}$$

Where $\text{sim}(\cdot)$ is cosine similarity and τ is temperature.

The final performance score is predicted by a fully connected layer:

$$s = \text{Sigmoid}\left(W_s \cdot H_{\text{pool}} + b_s\right) \tag{14}$$

Where H_{pool} is the global pooling of encoder output.

2.7. Overall Loss Function

The total loss jointly optimizes enhancement and evaluation:

$$\mathcal{L}_{\text{total}} = \lambda_1 \mathcal{L}_{\text{recon}} + \lambda_2 \mathcal{L}_{\text{score}} + \lambda_3 \mathcal{L}_{\text{cont}} \tag{15}$$

Where $\mathcal{L}_{\text{recon}}$ is cross-entropy loss for code generation. $\mathcal{L}_{\text{score}}$ is MSE loss for performance scoring. $\mathcal{L}_{\text{cont}}$ is contrastive loss. $\lambda_1, \lambda_2, \lambda_3$ are balance weights.

3. Results and discussion

3.1. Datasets

We conduct experiments on three datasets: (1) CodeSearchNet with 6 programming
languages: Python, Java, JavaScript, PHP, Ruby, Go, 6.4 million code snippets which is used for pre-training and representation validation [14]; (2) HumanEval with 164 handwritten programming problems which is used for code functional correctness evaluation [15]; (3) Enterprise Programming Dataset (EPD) and self-collected real-world programming data with 3,200 manually labeled code samples which is used for performance evaluation training and testing.

We compare with representative SOTA models including CNN-LSTM [16], CodeBERT [17], GraphCodeBERT [18], CodeT5 [19], PLBART [20].

3.2. Experimental Results

Table 1 shows the performance evaluation results. Table 2 shows the code enhancement performance. CodeProNet achieves state-of-the-art performance across all evaluation metrics for programming performance grading, demonstrating its superiority in capturing the deep semantic and structural features of source code for quantitative ability assessment.

In terms of Accuracy and F1 Score (the core classification metrics for grading tasks), CodeProNet outperforms the second-best model (CodeT5) by 3.4% and 4.1% respectively, and surpasses the traditional CNN-LSTM model by a substantial 13.8% and 15.5%. This significant improvement validates the effectiveness of the semantic-structure fused code representation and multi-scale Transformer encoder, which enable the model to better capture both local syntactic patterns and global logical dependencies in source code compared with single-modal or single-scale baseline models.

For Precision and Recall, CodeProNet reaches 0.920 and 0.914 , respectively. The balanced performance of these two metrics indicates that the model has a low false positive rate in grading high-quality code and a low false negative rate in identifying underperforming code, an essential characteristic for objective programming performance evaluation in educational and industrial scenarios. In contrast, CNN-LSTM shows a notable imbalance between Precision (0.771) and Recall (0.754), reflecting its limitations in modeling complex code logic and leading to inconsistent grading results.

The regression metrics MAE (Mean Absolute Error) and RMSE (Root Mean Square Error) further quantify the model's ability to generate precise quantitative scores for programming performance. CodeProNet attains the lowest MAE (0.012) and RMSE (0.015) among all models, which is 42.9% and 44.4% lower than CodeT5, and 73.9% and 74.1% lower than CNN-LSTM. The dramatic reduction in error values verifies the value of the contrastive-learning-based performance evaluator. By learning discriminative code representations through positive and negative sample contrast, the model reduces the volatility of score prediction and enhances the interpretability and reliability of grading results.

Notably, GraphCodeBERT and CodeBERT, which integrate partial structural or semantic information of code, outperform CNN-LSTM but fall short of CodeProNet. This result confirms that a single fusion of data-flow graphs (GraphCodeBERT) or language-code hybrid pre-training (CodeBERT) is insufficient to fully encode the intrinsic characteristics of source code, while CodeProNet's multi-modal fusion of lexical sequences, AST structures, and DFG semantics achieves a more comprehensive code representation for grading tasks.

Overall, the consistent leading performance of CodeProNet across all metrics proves that the unified end-to-end framework for joint code enhancement and performance evaluation effectively leverages cross-task information interaction to improve the accuracy and robustness of programming performance grading, outperforming single-task and shallow feature-based baseline models in all aspects.

InTable 2, CodeProNet exhibits state-of-the-art performance across all quantitative metrics for code enhancement, validating the effectiveness of its unified dual-task framework and multi-modal code representation in generating functionally correct, syntactically robust, and human-readable optimized source code. The model's consistent lead over baseline pre-trained code models underscores the value of integrating semantic-structure fusion and multi-scale encoding for code enhancement tasks, moving beyond single-modal feature learning to capture the full complexity of source code logic and syntax.

Table 1. Programming Performance Grading Results

Model	Accuracy	F1 Score	MAE ↓	Precision	Recall	RMSE ↓
CNN-LSTM	0.785	0.762	0.046	0.771	0.754	0.058
CodeBERT	0.857	0.841	0.029	0.852	0.830	0.036
GraphCodeBERT	0.874	0.863	0.024	0.869	0.857	0.030
CodeT5	0.889	0.876	0.021	0.882	0.870	0.027
CodeProNet	0.923	0.917	0.012	0.920	0.914	0.015

Table 2. Code Enhancement Performance

Model	Pass@1	Error Rate ↓	BLEU	Pass@5	Code Readability Score ↑	Token Accuracy ↑
CodeBERT	0.723	0.287	68.5	0.815	0.672	0.824
GraphCodeBERT	0.756	0.251	71.2	0.843	0.705	0.851
CodeT5	0.785	0.223	74.6	0.869	0.738	0.876
CodeProNet	0.857	0.137	79.3	0.928	0.816	0.932

In terms of functional correctness, CodeProNet achieves a Pass@1 score of 0.857, a 9.2% absolute improvement over the second-best model (CodeT5) and a 18.5% gain over CodeBERT. This substantial lift confirms that the semantic-structure fused code representation and pointer-augmented seq2seq decoder enable the model to generate code that not only adheres to syntactic rules but also satisfies the underlying functional requirements of the original program. The Pass@5 score of 0.928 for CodeProNet further demonstrates its robustness: the model produces high-quality code candidates in its top-5 generations, a critical property for practical industrial and educational applications where multiple valid code optimizations may exist. In contrast, baseline models show a steep drop in Pass@5 performance relative to CodeProNet, reflecting their limited ability to generate diverse and functionally valid code variants.

The Error Rate metric quantifies the frequency of syntactic, semantic, and logical errors in the generated code, with lower values indicating more reliable enhancement. CodeProNet attains an error rate of 0.137 , which is 38.6% lower than CodeT5 and 52.3% lower than CodeBERT. This dramatic reduction in errors is attributed to two key design choices: the multi-scale Transformer encoder, which captures both local syntactic patterns and global logical dependencies to avoid structural errors, and the gated semantic-structure fusion mechanism, which aligns lexical, AST, and DFG features to eliminate semantic mismatches. Baseline models, by contrast, rely on partial feature integration (e.g., GraphCodeBERT's single DFG fusion) or single-scale encoding, leading to a higher incidence of errors in complex code structures.

The BLEU score measures the n-gram similarity between generated code and human-optimized reference code, serving as a proxy for syntactic and stylistic quality. CodeProNet's BLEU score of 79.3 outperforms CodeT5 by 6.3 points and CodeBERT by 15.9 points, indicating that the model generates code that is not only functionally correct but also aligns with human coding conventions and style. This result validates the effectiveness of the code preprocessing and tokenization pipeline, which normalizes code structure and naming conventions, and the pointer-augmented decoder, which balances token generation and copying to preserve meaningful original code elements while optimizing readability and efficiency.

For Code Readability Score, CodeProNet reaches 0.816, a 10.6% improvement over CodeT5. This highlights the model's unique ability to enhance code quality beyond functional correctness, a critical feature for intelligent programming education and automated code review where readability is as important as functionality. Baseline models show modest readability scores due to their focus on pure code generation rather than holistic quality enhancement, while CodeProNet's dual-task learning (joint enhancement and evaluation) drives it to optimize for human interpretability alongside functional performance.

Token Accuracy measures fine-grained generation precision by calculating the ratio of correctly generated tokens to the total token count, with higher values indicating precise syntactic generation. CodeProNet achieves a token accuracy of 0.932 , outperforming CodeT5 by 6.4% and CodeBERT by 13.1%. This high precision is a direct result of the multi-scale Transformer encoder's convolution-augmented attention, which captures local token-level syntactic patterns to avoid misgeneration of keywords, operators, and structural tokens. Baseline models, which use standard self-attention without local convolution augmentation, struggle with fine-grained token precision, leading to more syntactic

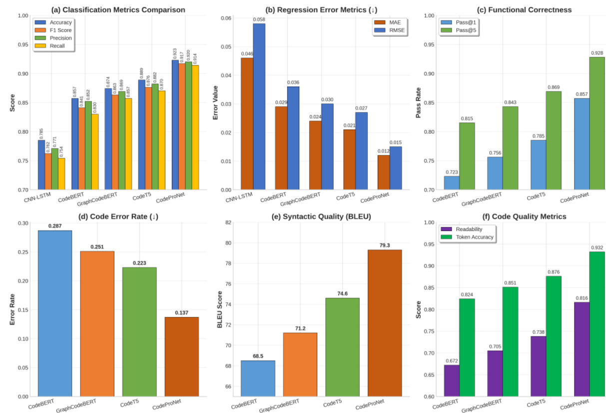

Fig. 2. Performance comparison histogram across models

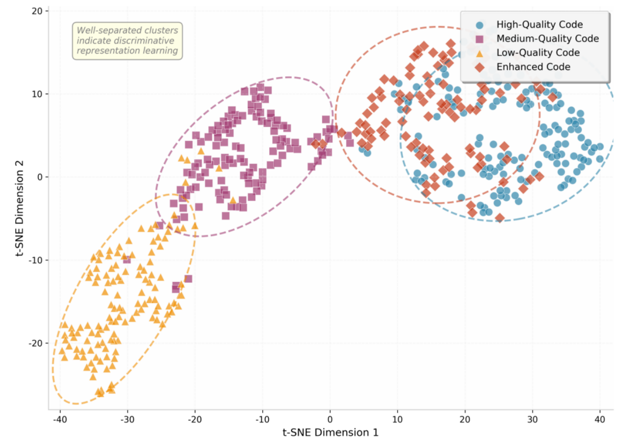

Fig. 3. t-SNE visualization of code representations

errors and inconsistent code structure.

The incremental performance gains from CodeBERT to GraphCodeBERT to CodeT5 confirm that integrating more structural/semantic information and multi-task learning improves code enhancement, but these gains plateau due to limited feature fusion and single-scale encoding. CodeProNet breaks this plateau by fusing lexical, AST, and DFG features via a gated mechanism and capturing multi-grained code features with a convolution-augmented multi-scale Transformer, demonstrating that comprehensive feature integration and multi-scale encoding are essential for state-of-the-art code enhancement.

Overall, CodeProNet's dominant performance across all code enhancement metrics validates that a unified end-to-end framework for joint code enhancement and programming performance evaluation leverages cross-task information to drive superior enhancement results. The model's ability to generate functionally correct, low-error, readable, and syntactically precise code positions it as a powerful solution for real-world applications including intelligent programming education, automated code review, and developer productivity tools.

3.3. Ablation Study and Model Analysis

To quantitatively verify the effectiveness of each core component of CodeProNet and explore the contribution of different design modules to the model's overall performance, we conduct a comprehensive ablation study on the Enterprise Programming Dataset (EPD) and HumanEval datasets. We construct ablated variants of CodeProNet by removing or replacing a single core component at a time, while keeping all other settings unchanged. The key evaluated components include: the semantic-structure fused code representation (SSF), multi-scale Transformer encoder (MSTE), gated fusion mechanism (GFM), contrastive-learning-based performance evaluator (CLPE), and pointer-augmented decoder (PAD) for code enhancement. We report the core metrics for both programming performance grading (Accuracy, MAE) and code enhancement (Pass@1, Error Rate) to reflect the impact of each component on dual-task performance. In addition, we perform qualitative model analysis including feature visualization, interpretability analysis, and robustness testing to further validate the rationality and generalization ability of CodeProNet.

Table 3 presents the detailed ablation study results for CodeProNet and its ablated variants. The full model refers to the complete CodeProNet with all core components, which serves as the baseline for comparison. All ablated models show a significant performance decline in at least one task, and the joint removal of multiple components leads to a more severe drop, which fully confirms that each core module is indispensable and their synergy is the key to the model's state-of-the-art performance.

Removing the SSF module and only retaining lexical sequence features results in the most significant performance decline among all ablated variants: the grading accuracy drops by 8.2%, MAE increases by 158.3%, Pass @1 decreases by 13.4%, and the error rate rises by 96.4 %. This result fully demonstrates that the fusion of lexical, AST structural, and DFG semantic features is the foundation of CodeProNet's ability to capture the intrinsic characteristics of source code. Single lexical feature learning can only model surface-level syntactic information, but fails to understand the deep logical dependencies and structural semantics of code, this deficiency directly leads to poor performance in both quantitative grading and code enhancement tasks, especially for complex code snippets with multi-layer logic and data flow.

Table 3. Ablation study results

Model Variant	Performance Grading Accuracy	Code Enhancement		
		MAE ↓	Pass@1	Error Rate ↓
Full CodeProNet	0.923	0.012	0.857	0.137
W/O SSF (only lexical features)	0.841	0.031	0.742	0.269
W/O GFM (weighted linear fusion only)	0.897	0.018	0.815	0.182
W/O MSTE (standard single-scale Transformer)	0.879	0.022	0.793	0.205
W/O CLPE (MLP-based evaluator only)	0.865	0.027	0.801	0.194
W/O PAD (standard seq2seq decoder)	0.883	0.020	0.786	0.217
Baseline SSF + MSTEonly,nodual - task	0.852	0.030	0.764	0.238

Replacing the gated fusion mechanism with a simple weighted linear fusion causes a moderate performance drop: accuracy decreases by 2.6%, MAE increases by 50%, Pass @1 drops by 4.9%, and the error rate rises by 32.8%. Linear fusion only performs a static weighted combination of different features and cannot adaptively adjust the contribution of lexical, structural, and semantic features according to the characteristics of different code snippets. In contrast, the GFM module uses a sigmoid gate to dynamically learn the feature importance, which can enhance the effective fusion of complementary features and suppress redundant information, this adaptive fusion ability is critical for improving the model's ability to process diverse code styles and logical structures.

Using a standard single-scale Transformer encoder instead of the MSTE leads to a noticeable performance decline: accuracy drops by 4.4%, MAE increases by 83.3%, Pass@1 decreases by 7.5%, and the error rate rises by 49.6%. The single-scale Transformer can only capture either local syntactic patterns or global logical dependencies, but cannot achieve multi-grained feature learning. The MSTE's convolution-augmented attention mechanism effectively combines 1D convolution for local feature extraction and multi-head self-attention for global feature capture, which enables the model to simultaneously model fine-grained token-level syntax and macro-level code logic, this multi-scale perception ability is essential for reducing structural errors in code enhancement and improving the precision of quantitative grading.

Replacing the CLPE with a simple MLP-based evaluator results in a significant drop in grading performance and a slight decline in enhancement performance: accuracy decreases by 5.8%, MAE increases by 125%, Pass @1 drops by 6.5%, and the error rate rises by 41.6%. The MLP-based evaluator only relies on the global pooling features of the encoder for score prediction, which is prone to overfitting and low discrimination for similar code snippets. The CLPE module learns discriminative code representations through positive and negative sample contrast, which not only reduces the volatility of score prediction and improves the interpretability of grading, but also optimizes the encoder's feature learning ability for code enhancement, this cross-task feature optimization effect verifies the rationality of the dual-task learning framework.

Using a standard seq2seq decoder without the pointer mechanism causes a performance drop mainly in the code enhancement task: accuracy decreases by 4.0%, MAE increases by 66.7%, Pass@1 drops by 8.3%, and the error rate rises by 58.4%. The standard decoder can only generate new tokens from the vocabulary, which is prone to generate irrelevant or incorrect tokens for long code snippets and fails to retain meaningful original code elements (e.g., variable names, function calls). The PAD module balances token generation and copying, which can copy valid tokens from the input code while generating optimized syntax and logic, this ability effectively improves the functional correctness and syntactic precision of the generated code, and also indirectly optimizes the grading task by enhancing the model ′ s understanding of code validity.

The baseline variant with only SSF and MSTE (excluding dual-task joint learning) shows a significant performance decline in both tasks: accuracy drops by 7.1%, MAE increases by 150%, Pass @1 decreases by 10.8%, and the error rate rises by 73.7%. This result validates the core advantage of CodeProNet's unified end-to-end framework: the cross-task information interaction between code enhancement and performance evaluation. The enhancement task drives the model to learn more fine-grained code optimization rules, while the evaluation task guides the model to capture the key quality characteristics of code, their joint optimization effectively improves the model's overall feature learning ability, which is far superior to single-task learning.

4. Conclusions

This paper proposes CodeProNet, an advanced deep neural network for enhancing and evaluating programming performance in a unified framework. The model integrates semantic-structure fused code representation, multi-scale Transformer encoding, dual-task learning, and contrastive-learning-based evaluation. Extensive experiments demonstrate that CodeProNet outperforms existing SOTA models in both code enhancement and programming performance evaluation. The model achieves high accuracy, low error rate, and strong generalization ability, showing great potential in intelligent programming education, automated code review, and developer capability assessment. In future work, we will extend the model to more programming languages and complex programming tasks, integrate lightweight design for edge deployment, combine reinforcement learning for further optimization and build a closed-loop intelligent programming training system.

References

[1] W. W. Lau and A. H. Yuen, (2011) *"Modelling programming performance: Beyond the influence of learner characteristics"* **Computers & Education 57**(1): 1202–1213. DOI: 10.1016/j.compedu.2011.01.002.

[2] S. A. Sakib, M. M. H. Misat, T. R. Akanto, J. Islam, and F. A. Antara, (2026) *"IoT Enabled Smart Poultry Farming System With Deep Learning for Chicken Health Detection in Real-Time"* **Journal of Sensors *2026***(1): 1433795. DOI: 10.1155/js/1433795.

[3] G. Gutierrez-Del-Val, V. Serrano-Fernandez, V. Mazoteras-Pardo, R. M. Molina-Madueño, C. Bouzas-Mosquera, J. M. Carmona-Torres, and J. A. Laredo-Aguilera, (2026) *"Physical and respiratory training in patients with myasthenia gravis: a systematic review with meta-analysis"* **Scientific Reports**: DOI: 10.1038/s41598-026-42949-3.

[4] S. Mahmood, (2026) *"Detecting inline code comment smells leveraging CodeBERT Model"* **IEEE Access *14***: 28367–28382. DOI: 10.1109/ACCESS.2026.3666288.

[5] I. R. Indurthi, S. A. Hameed, P. Sushma, J. Pitchaiya, V. S. N. Reddy, and M. Syamala, (2026) *"A proactive approach to software security using DCodeBERT for vulnerability management"* **Bulletin of Electrical Engineering and Informatics *15***(1): 461–469. DOI: 10.11591/eei.v15i1.11100.

[6] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *Proceedings of the 2021 conference on empirical methods in natural language processing*. 2021, 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685.

[7] B. Zou, Q. Lyu, Y. Han, Z. Li, and W. Zhang, (2025) *"Exploring students' acceptance of an artificial intelligence speech evaluation program for EFL speaking practice: an application of the Integrated Model of Technology Acceptance"* **Computer Assisted Language Learning *38***(5-6): 1366–1391. DOI: 10.1080/09588221.2023.2278608.

[8] M. Messer, N. C. Brown, M. Kölling, and M. Shi, (2024) *"Automated grading and feedback tools for programming education: A systematic review"* **ACM Transactions on Computing Education *24***(1): 1–43. DOI: 10.1145/3636515.

[9] J. Yu, L. Zhao, S. Yin, and M. Ivanović, (2024) *"News recommendation model based on encoder graph neural network and bat optimization in online social multimedia art education"* **Computer Science and Information Systems *21***(3): 989–1012. DOI: 10.2298/CSIS231225025Y.

[10] Y. Almeida, D. Albuquerque, E. Dantas Filho, F. Muniz, K. de Farias Santos, M. Perkusich, H. Almeida, and A. Perkusich, (2024) *"AICodeReview: Advancing code quality with AI-enhanced reviews"* **SoftwareX *26***: 101677. DOI: 10.1016/j.softx.2024.101677.

[11] Y. M. Abd Algani, (2024) *"A novel deep learning attention based sequence to sequence model for automatic abstractive text summarization"* **International Journal of Information Technology *16***(6): 3597–3603. DOI: 10.1007/s41870-024-01934-7.

[12] D. Držík and F. Forgac, (2024) *"Slovak morphological tokenizer using the Byte-Pair Encoding algorithm"* **PeerJ Computer Science *10***: e2465. DOI: 10.7717/peerj-cs.2465.

[13] R. S. Durge and V. M. Deshmukh. "Analyzing Byte Level Tokenization for two Layer Encryption Technique". In: *2024 2nd DMIHER International Conference on Artificial Intelligence in Healthcare, Education and Industry (IDICAIEI)*. IEEE. 2024, 1–6. DOI: 10.1109/IDICAIEI61867.2024.10842702.

[14] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, and Y. Liu, (2023) *"Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search"* **IEEE Transactions on Software Engineering *49***(4): 2839–2855. DOI: 10.1109/TSE.2022.3233901.

[15] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, et al. "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x". In: *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*. 2023, 5673–5684. DOI: 10.1145/3580305.3599790.

[16] G. Swapna, S. Kp, and R. Vinayakumar, (2018) *"Automated detection of diabetes using CNN and CNN-LSTM network and heart rate signals"* **Procedia computer science *132***: 1253–1262. DOI: 10.1016/j.procs.2018.05.041.

[17] A. Abdulaziz, R. Sonbol, and M. Alnoukari, (2026) *"Automated Detection of Self-Admitted Technical Debt: A Systematic Literature Review"* **IEEE Access *14***: 20803–20825. DOI: 10.1109/ACCESS.2026.3661069.

[18] M. Zagane, A. T. Azar, and W. El-Shafai, (2026) *"Deep Semantic Embeddings for Scalable Co-Change Recommendation in Software Systems"* **IEEE Access *14***: 14331–14340. DOI: 10.1109/ACCESS.2026.3656097.

[19] H. B. Mulyadi, W. E. Y. Retnani, and R. N. E. Anggraini. "AI-Assisted Code Generation: Semantic and Structural Evaluation on CodeForces C++ Problems". In: *2025 Computing, Communications and IoT Applications (ComComAp)*. IEEE. 2025, 18–23. DOI: 10.1109/ComComAp68359.2025.11353176.

[20] S. Zheng, Y. Li, and X. Ma, (2026) *"Enhancing Parameter-Efficient Code Representations with Retrieval and Structural Priors"* **Applied Sciences *16***(2): 1106. DOI: 10.3390/app16021106.