Bidirectional Long Short-Term Memory Networks for Automated Source Code Generation

F.D. Nyaga

National University of Science and Technology "MISiS" (Moscow)

Abstract: This paper examines the application of Bidirectional Long Short-Term Memory (Bi-LSTM) networks in neural source code generation. The research analyses how Bi-LSTMs process sequential data bidirectionally, capturing contextual information from both past and future tokens to generate syntactically correct and semantically coherent code. A comprehensive analysis of model architectures is presented, including embedding mechanisms, network configurations, and output layers. The study details data preparation processes, focusing on tokenization techniques that balance vocabulary size with domain-specific terminology handling. Training methodologies, optimization algorithms, and evaluation metrics are discussed with comparative results across multiple programming languages. Despite promising outcomes, challenges remain in functional correctness and complex code structure generation. Future research directions include attention mechanisms, innovative architectures, and advanced training procedures.

Keywords: code generation, deep learning, recurrent neural networks, transformers, tokenisation.

Introduction

Neural source code generation represents a paradigm shift in software development, moving away from traditional rule-based programming towards data-driven approaches that leverage the power of deep learning models to automatically generate source code from various inputs, such as natural language descriptions or abstract specifications [1, 2]. This emerging field holds immense potential to revolutionise software engineering practices by automating repetitive coding tasks, accelerating development cycles, and empowering individuals with limited programming expertise to create software applications [2]. Among the diverse range of deep learning architectures employed in neural source code generation, Bidirectional Long Short-Term Memory (Bi-LSTM) networks have emerged as a prominent and effective technique [3, 4].

Bidirectional long short-term memory networks are among the most popular and powerful deep learning methods in neural source code generation. Bi-LSTMs leverage their ability to handle bidirectional sequential data streams to generate source code; this allows them to easily process the intricate dependencies and contextual nuances found in programming languages [5].

Bi-LSTMs improve code structure comprehension by incorporating context from both past and future tokens, as opposed to unidirectional LSTMs, which incorporate only past context. This is especially crucial for ensuring the syntactic correctness and semantic coherence of the generated code[5, 6]. Bi-LSTM-based models are gaining popularity as demand for speedy and automated code generation tools rises. They provide an appealing path for software development process advancement and unlock new potential for both novice and professional developers. Furthermore, deep learning innovations have given rise to code-generation models capable of producing highly accurate source code from code-based and natural language requests.

This paper presents an overview of bidirectional LSTM-based techniques for automated source code generation, including their structures, training procedures, and applications.

Background

Improved performance of large language models has significantly helped natural language processing by bridging the gap between programming and natural languages. AI-based code generation produces source code from natural language descriptions, improving efficiency. Although early solutions relied on heuristic rules and expert systems, recent breakthroughs in deep learning, such as recurrent neural networks and transformers, have proven particularly beneficial in addressing code production challenges [6].

Bi-LSTM Networks

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural network developed primarily to address the issue of vanishing gradients in traditional RNNs during the handling of long sequences. They consist of memory cells with input, forget, and output gates to regulate the flow of information. This allows for the establishment of long-term interdependence through the management of information intake, omitting of unnecessary information, and cell state contribution to outputs [7].

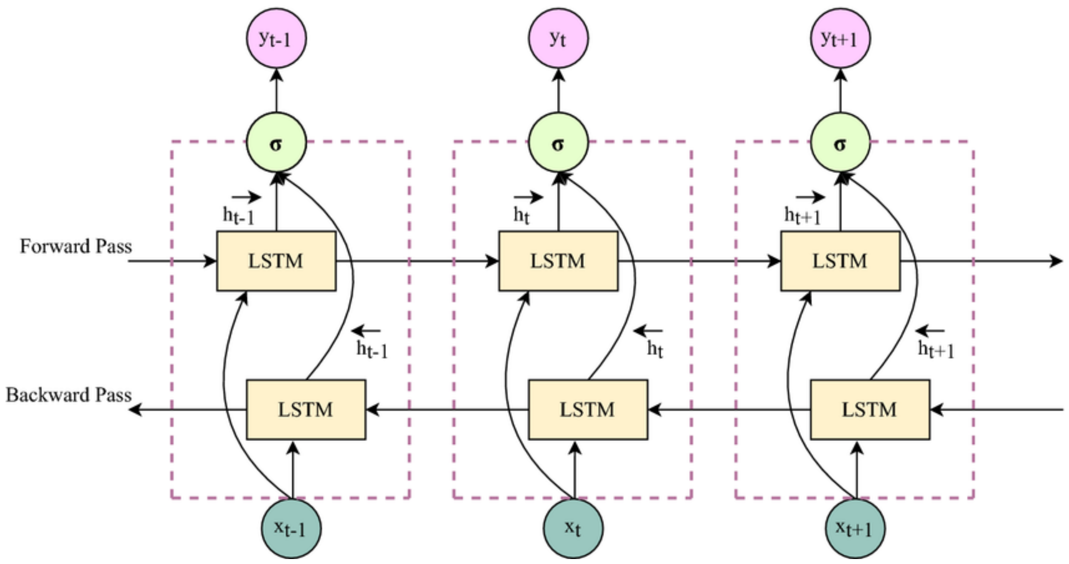

Fig.1. - Bidirectional LSTM Architecture

Furthermore, the recurrence of LSTMs retains previous inputs as hidden states, which is important for comprehending sequential context in tasks such as source code generation and natural language analysis. Bidirectional LSTMs improve the performance of standard LSTMs by capturing both forward and backward input sequences. This is highly useful for applications that need to provide context based on previous and future inputs (see Fig.1. for a visual illustration).

Neural Source Code Generation

The capability of generating code from diverse inputs, such as formal requirements, natural language descriptions, or existing code snippets, makes neural source code generation a significant software development innovation. Previous code generation techniques were based on hand-coded rules and templates that were rigid and difficult to generalize across diverse programming areas and application domains. Yet, the development of deep learning techniques has changed this industry and resulted in models that are able to generate code with greater accuracy and complexity [7 - 9].

Neural source code generation creates new and contextually relevant fragments of code by leveraging neural networks' ability to understand the complex links between input data and output code. These models can learn the syntax, semantics, and stylistic guidelines of several programming languages because they are trained on large code and documentation datasets [9, 10].

Translating natural language specifications into executable code has the potential to democratise software development since it will enable users without programming experience or knowledge to execute computational ideas and share in the digital world, making the technological ecosystem more diverse and inclusive [3, 22]. Neural code generation can also dramatically enhance developer productivity by automating routine coding tasks and facilitating more creative and strategic work, during the software development lifecycle [10].

Tokenization and Data Preparation

Adequate data preparation is critical when building Bi-LSTM-based models because it has a substantial impact on model development and training capabilities. Tokenisation is the initial stage of this approach, in which the source code is broken down into discrete tokens. Tokens are the fundamental building elements of code and can include operators, punctuation, keywords, and identifiers. Because it accurately tokenises the code, the model can grasp and evaluate its underlying structure and semantics, allowing it to perform better on tasks like code generation and analysis [10 - 12].

Furthermore, techniques such as Subword tokenisation are employed to achieve a balance between the model's ability to handle uncommon or unknown words in source code—words with unusual naming patterns or that are domain-specific—and vocabulary size. This improves the realism and consistency of generated code by significantly increasing the model's ability to comprehend the intricate technical aspects prevalent in various programming paradigms and coding styles [9, 10].

After tokenising the source code, it is necessary to create a vocabulary in which each token is assigned a fixed-sized embedded vector. Keywords and semantically relevant elements should be expressed using similar vectors [13-16]; some systems train a token embedding first, followed by the neural network.

Bi-LSTM Model Architecture

The Bi-LSTM structure utilized for code generation typically comprises three layers: an embedding layer, Bi-LSTM layers, and a dense output layer. The embedding layer converts each token into a high-dimensional vector representation containing semantic and syntactic information regarding the token; the embeddings are passed into the Bi-LSTM layers. The layers process the input sequence in both forward and backwards directions, gathering contextual information from tokens that precede and follow each token and computing a hidden state representation for each token in the sequence [17].

The Bi-LSTM structure utilized for code generation typically comprises the hidden state representations that are sent into a dense output layer, which produces a probability distribution over the multiple token vocabularies. Hyperparameters with a significant impact on model performance include the number of Bi-LSTM layers, learning rate, batch size, and hidden state dimension. Furthermore, tuning them is essential for improving model performance. In addition, grid search and random search are common methods for determining the optimal hyperparameter configurations [17, 18].

Bi-LSTM models are extensively trained on large source code repositories utilizing optimisation techniques to reduce disparities between the generated code and the desired output. Throughout training, model parameters (weight and bias) are constantly changed to improve context sensitivity and code generation accuracy. Parameters are updated based on observations of the loss function gradient for prediction-target differences. Adam, RMSprop optimisation, and stochastic gradient descent are used to reduce loss. The Adam optimiser uses gradient information to continually alter learning rates for each parameter [19].

To support thorough evaluation of a variety of programming languages, the models are extensively tested on training data sets representing a variety of coding styles, domains, and complexities, as well as a variety of programming languages (C, C++, Java, and Python). Various metrics are used to evaluate performance: for example, BLEU which uses n-gram to assess the similarity between reference and output code [20 - 24].

Conclusion

This study examined the potential of Bi-LSTM networks to generate source code, as well as the benefits and drawbacks of this technique. Despite the encouraging results of these approaches, future research should focus on improving functional correctness and handling complicated code structures. Potential research paths include incorporating attention mechanisms, studying innovative network designs, and developing increasingly sophisticated training procedures.

Литература/References

1. Zhang X., et al. Context-aware code generation with synchronous bidirectional decoder. Journal of Systems and Software. 2024. Vol. 214. P. 112066.
2. Li J., et al. Large language model-aware in-context learning for code generation. ACM Transactions on Software Engineering and Methodology. 2023.
3. Wang Y., et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859. 2021.
4. Svyatkovskiy A., et al. Intellicode compose: Code generation using transformer. Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2020.
5. Chen M., et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. 2021.
6. Li Y., et al. Competition-level code generation with alphacode. Science. 2022. Vol. 378. No. 6624. Pp. 1092-1097.
7. Hellendoorn V. J., Devanbu P. Are deep neural networks the best choice for modeling source code? Proceedings of the 2017 11th Joint meeting on foundations of software engineering. 2017. Pp. 763-773.
8. Allamanis M., Brockschmidt M., Khademi M. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740. 2017.

9. Alon U., et al. code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400. 2018.
10. Zhou Y., et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems. 2019. Vol. 32.
11. Feng Z. et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155. 2020.
12. Guo D., et al. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366. 2020.
13. Zhang J., et al. Retrieval-based neural source code summarization. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020.
14. Vaswani A. et al. Attention is all you need. Advances in neural information processing systems. 2017. Vol. 30.
15. Austin J., et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732. 2021.
16. Roziere B., et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950. 2023.
17. Zhou S., et al. Docprompting: Generating code by retrieving the docs. arXiv preprint arXiv: 2207.05987. 2022.
18. Dong Y., et al. Self-collaboration code generation via chatgpt. ACM Transactions on Software Engineering and Methodology. 2024. Vol. 33. No. 7. Pp. 1-38.
19. Li R. et al. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161. 2023.
20. Nath P., et al. AI and Blockchain-based source code vulnerability detection and prevention system for multiparty software development. Computers and Electrical Engineering. 2023. Vol. 106. P. 108607.

21. Li J., et al. Skcoder: A sketch-based approach for automatic code generation. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.
22. KC D., Morrison C. T. Neural machine translation for code generation. arXiv preprint arXiv:2305.13504. 2023.
23. Pradel M., Chandra S. Neural software analysis. Communications of the ACM. 2021. Vol. 65. No. 1. Pp. 86-96.
24. Van Houdt G., Mosquera C., Nápoles G. A review on the long short-term memory model. Artificial Intelligence Review. 2020. Vol. 53. No. 8. Pp. 5929-5955.

Дата поступления: 8.04.2025

Дата публикации: 25.05.2025