Evaluating Transformer-based Code Generation with NMT Transfer Learning: Comparison across Compiled and Interpreted Languages

Rithvik Sunku
University of California, Berkeley
rithviksunku@berkeley.edu

Savinay Chandrupatla
University of California, Berkeley
savinay@berkeley.edu

Abstract

This study delves into the effectiveness of Transformer-based models for generating code from text, focusing on scenarios with less structured data and a wider range of languages. We leverage transfer learning from pre-trained Natural Language Translation (NLT) models to enhance the performance of the code generation process. Our research explores the potential for achieving comparable or even superior performance compared to existing traditional approaches. Additionally, we analyze the impact of language type (compiled vs. interpreted) on the accuracy and robustness of code generation, providing valuable insights into the relationship between language characteristics and model performance.

1 Introduction

Deep learning has profoundly impacted the landscape of code generation, with tree-based approaches initially spearheading the movement. Recently, Transformer-based models have emerged as a powerful alternative to traditional approaches for code generation. While existing research primarily focuses on structured data like Abstract Syntax Trees (ASTs) and large code repositories, this study explores the potential of transformer-based approaches utilizing less structured data and a wider range of languages through NLT transfer learning (Junczys-Dowmunt et al., 2018). This study addresses the following research questions:

RQ1: Can Transformer models pre-trained on NLT data and fine-tuned on diverse programming datasets containing less structured data achieve comparable or improved performance compared to traditional approaches like AST-based models?

RQ2: Does extending the functionality of NLT models through transfer learning effectively generate accurate and robust code across various languages, including compiled and interpreted languages?

RQ3: Is there a significant difference in code generation performance between compiled and interpreted languages when using Transformer models with Natural Language Translation (NLT) transfer learning?

By employing a strategic dataset approach and analyzing the performance of Transformer models across different languages and data types, this study aims to:

- Validate the effectiveness of Transformer-based models with NLT transfer learning for text-to-code generation with less structured data.
- Identify the impact of data quality, language type (compiled vs. interpreted), and NLT pre-training on code generation accuracy and robustness.
- Contribute to the development of more robust and generalizable code generation models that can effectively handle diverse programming languages and data formats.

2 Background

2.1 Deep Learning for Code Generation

The field of code generation has witnessed a transformative shift with the advent of deep learning. Researchers have successfully harnessed the power of neural networks to automate various programming tasks, demonstrating remarkable potential for revolutionizing software development. Pioneering work such as (Le et al., 2020) demonstrated that deep learning models can achieve state-of-the-art results in generating general-purpose code.

2.2 Tree-based Models

Initially, tree-based models employing Abstract Syntax Trees (ASTs) emerged as a frontrunner. Works like Tree2Tree NMT (Chakraborty et al., 2020) and TreeGen (Sun et al., 2019) leveraged the ability of ASTs to capture the hierarchical structure and semantics of code, achieving impressive results. The success of these models highlighted the importance of explicitly modeling the structure of code for accurate and efficient generation or translation.

However, relying on ASTs comes with limitations. Their construction and manipulation require significant computational resources, making them less efficient compared to other approaches. Additionally, the performance of AST-based models can be sensitive to changes in the structure and format of the input code.

2.3 Advances in Natural Language and Transformer-based Models

Recognizing the limitations of ASTs, research efforts shifted towards utilizing natural language (NL) and less structured data as input for code generation. Works like the Syntactic Neural Model (Yin and Neubig, 2017) demonstrated the ability to achieve competitive performance using only NL descriptions, suggesting that the inherent grammatical structure of code could be effectively learned from natural language alone.

Further advancements came with the introduction of Transfer Learning and the Transformer architecture (Vaswani et al., 2023). Code2Seq (Alon et al., 2019) showcased the potential of combining NL and code data for improved AST creation, while CodeBERT (Zhang et al., 2023) demonstrated that pre-trained models can achieve comparable or even better performance than RNN-based approaches. Additionally, works like MarianCG (Soliman et al., 2022) and CodeParrot (Sharma et al., 2023) highlighted the effectiveness of extending NMT models and Transformer architectures to achieve state-of-the-art results in code generation (KC and Morrison, 2023).

2.4 Impact of Different Coding Languages

While existing research demonstrates the effectiveness of Deep Learning for code generation, a critical gap remains in our understanding of the robustness of these models across diverse coding languages (Norouzi et al., 2021). Analysis of coding languages clarify that languages are very different and can't be treated the same way in terms of portability, readability, and expressiveness (Alomari et al., 2015). Research like ReCode (Wang et al., 2022) suggests that different types of languages exhibit distinct characteristics and may respond differently to various code generation models. Additionally, studies like CodeAttack (Jha and Reddy, 2023) suggest that compiled languages might be less susceptible to attacks compared to interpreted languages. This highlights the need for further investigation into the impact of language type on code generation performance.

2.5 Reliability and Explainability of Code Generation Models

Recent work by (Liu et al., 2023) emphasizes the importance of evaluating the robustness and reliability of code generation models, particularly when utilizing less structured data. This motivates the need for research that assesses the performance of different models across various languages and data types, aiming to develop robust and generalizable code generation models.

This background section provides a comprehensive overview of the existing literature on deep learning for code generation. It highlights the major advancements from tree-based models to Transformer-based approaches and underscores the importance of considering language type and data quality when evaluating code generation performance. This context sets the stage for our study, which aims to investigate the effectiveness of Transformer-based models for text-to-code generation with less structured data and a broader range of languages.

3 Methods

3.1 Overview

This study investigates the impact of fine-tuning the MarianMT NMT model on diverse programming languages, focusing on its effect on code generation performance as measured by BLEU and ROUGE scores.

3.2 Datasets

We utilize two publicly available datasets for text-to-code generation: CoNaLa: This benchmark dataset provides real-world programming examples sourced from Stack Overflow, offering a curated collection of natural language descriptions and corresponding code snippets. Each entry includes a

natural language description ("intent") and the corresponding code snippet ("snippet"). We exclude the "rewritten intent" column to maintain the original intent phrasing and avoid potential bias.

XLCoST: This comprehensive benchmark dataset contains text-to-code pairs for seven programming languages: Python, C, C#, C++, Java, Javascript, and PHP. It offers both snippet-level and program-level annotations, allowing us to analyze the model's performance across different granularities and programming paradigms. Due to limited computational resources, a subset of data from XLCoST is used for fine-tuning.

CoNaLa is chosen for its real-world examples and curated nature, providing a reliable benchmark for evaluating the model's ability to handle practical programming tasks. XLCoST complements this with its comprehensive set of diverse languages, enabling analysis of the model's performance across different paradigms and providing additional data for fine-tuning.

3.3 Data Preparation

The following steps are employed to prepare the data for fine-tuning:

- Tokenization: Both natural language descriptions and code snippets are tokenized using the Hugging Face tokenizer, ensuring consistency in data representation.
- Filtering: Irrelevant or noisy data is removed from both datasets to ensure training on high-quality examples.
- Subsetting: Due to computational resource limitations, a subset of data from XLCoST is selected for fine-tuning. This allows for efficient training and evaluation within the available resources.
- Mapping: A custom function `map_to_encoder_decoder_inputs` transforms the natural language descriptions and code snippets into appropriate inputs for the model. This includes encoding, padding, truncation, and label masking.
- Batching: The prepared data is batched with a size of `batch_size = 1` due to computational limitations.

The chosen tokenization parameters and padding/truncation strategies balance the need for efficient training with the accuracy of the generated code. The `map_to_encoder_decoder_inputs` function simplifies data preparation and ensures consistency across different programming languages. The complex list comprehension for label masking ensures accurate training by excluding padding tokens. The chosen batch size is a compromise between efficient training and resource limitations.

3.4 Model Architecture and Fine-tuning Procedure

The MarianMT NMT model, a state-of-the-art architecture for NLT is well-suited for text-to-code generation, is used as the base architecture for this study. Fine-tuning involves the following steps:

Pre-training: The model is pre-trained on a large corpus of natural language data to enhance its understanding of natural language semantics. Fine-tuning: The pre-trained model is then trained on the prepared subset of coding language-related data for each specific programming language (a combination targeting a mix of compiler and interpreted). Due to computational limitations, they were done in sequentially instead of in a typical batch. This fine-tuning adjusts the model parameters to better capture the syntax and semantics of each language combination.

Here are the specific language pairings we used and our reasons for the combinations:

- Python: This serves as the baseline model (CoNaLa and XLCoST) and represents a high-level language closest to natural language.
- Python, C++: This combination explores the impact of including a lower-level language (C++) on performance, potentially introducing more complex syntax and technical jargon. C++ was chosen due to its popularity and influence on other lower-level languages.
- Python, C: This combination investigates the effect of a low-level language with even greater distance from natural language, further challenging the model's adaptation. C was chosen as the foundation for many other languages, offering insights into adapting to low-level paradigms.
- Python, C++, Java: This combination incorporates both interpreted and compiled languages, providing insights into the model's ability

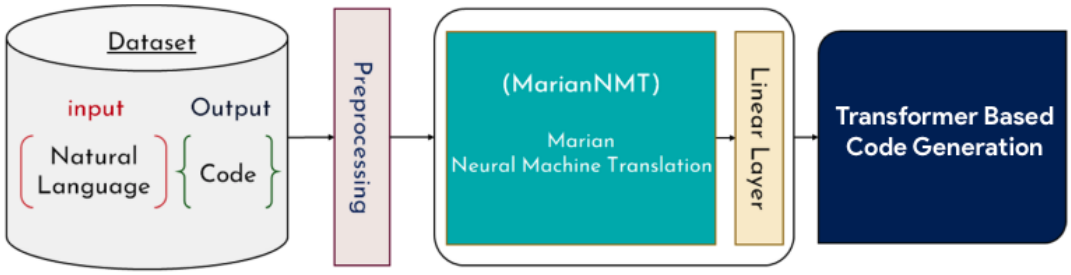

Fig. 1 NMT Transfer Learning for Transformer Based Code Generation (Adapted from MarianCG)

to handle diverse programming paradigms. Java was chosen as a widely used language with both interpreted and compiled features, adding another layer of complexity.

Fine-tuning allows the model to adapt to the specific characteristics of each programming language combination, leading to improved code generation accuracy and robustness.

3.5 Metrics

The performance of fine-tuned models is evaluated using two widely used metrics:

BLEU: This metric measures the similarity between generated code and reference code based on n-gram overlaps, assessing the accuracy of the generated code.

ROUGE: This metric evaluates the fluency and informativeness of generated code, considering word-level and phrase-level matches, assessing the robustness and quality of the generated code.

3.6 Evaluation

Due to computational limitations, we evaluate the fine-tuned models on a subset of the XLCoST dataset. Additionally, the CoNaLa dataset is used as a proxy to assess the performance of the MarianMT NMT model. This approach allows for a comprehensive analysis while considering computational constraints.

4 Results and Discussion

4.1 Overview

The following metrics were used to evaluate the performance of the machine translation models:

- **Bleu:** This metric measures the n-gram precision between the machine translation output and the reference translation. Higher Bleu scores indicate better performance.
- **Sacrebleu:** This metric is similar to Bleu, but it also takes into account the word order and fluency of the translation.
- **ROUGE-L:** This metric measures the recall of n-grams in the machine translation output compared to the reference translation. Higher Rouge-L scores indicate better performance.
- **Code Bleu:** This metric is specifically designed for evaluating code generation. It measures the similarity between the generated code and the reference code on various levels, including n-grams, syntax, and dataflow.
- **Exact Match:** This metric measures the percentage of times the generated code exactly matches the reference code.

The training of the models was very plateaued over the epochs, with minimal improvements occurring after 5 epochs. This suggests that 5 epochs were sufficient given computational constraints to achieve optimal performance.

4.2 Discussion of Training and Validation Results

The fine-tuning experiments revealed a clear trend of performance improvement as the model was exposed to increasingly diverse training data. The Python-only model significantly outperformed the CoNala (Python) baseline across key metrics, demonstrating the substantial benefit of domain-specific data augmentation. Adding C data further enhanced performance, showcasing the model's ability to adapt to different languages and benefit from cross-language knowledge transfer. This

Table 1: Code Generation Model Fine-tuning Results

Task	Epoch	Training	Validation	Bleu	Sacrebleu	Bleu-bigram	Rouge-2	Rouge-l	Sacre-bigram
CoNala (Python)	1	2.374	2.175	0.763	0.691	1.311	1.754	10.894	1.198
CoNala (Python)	2	1.854	1.944	1.714	1.506	2.590	2.723	11.197	2.188
CoNala (Python)	3	1.744	1.845	3.113	2.778	4.493	3.702	13.665	3.811
CoNala (Python)	4	1.600	1.798	3.371	3.072	4.825	3.377	14.439	4.241
CoNala (Python)	5	1.451	1.785	3.863	3.518	5.553	4.135	15.413	4.834
Python	1	1.246	1.302	15.370	22.504	17.366	28.075	45.543	25.932
Python	2	0.998	1.294	21.430	30.008	24.595	31.426	48.960	34.295
Python	3	0.929	1.271	20.471	29.508	26.467	31.883	48.407	36.820
Python	4	0.840	1.257	21.630	30.930	26.008	32.619	49.877	36.024
Python	5	0.770	1.259	21.472	30.608	25.664	31.630	49.422	35.668
Python + C	1	2.779	1.975	1.714	1.476	2.485	3.799	10.551	2.261
Python + C	2	1.870	1.800	3.792	3.319	4.268	9.795	14.781	3.844
Python + C	3	1.593	1.746	6.895	6.137	8.111	10.988	20.485	7.301
Python + C	4	1.467	1.720	7.457	6.820	8.552	10.716	21.888	7.984
Python + C	5	1.410	1.713	6.158	5.549	7.227	10.901	21.296	6.645
Python + C++	1	1.914	1.734	3.572	2.991	4.818	12.610	23.966	4.193
Python + C++	2	1.220	1.504	17.701	17.119	18.584	25.206	39.386	18.028
Python + C++	3	1.032	1.453	18.511	17.723	19.305	25.251	37.917	18.539
Python + C++	4	0.959	1.438	21.959	21.202	23.174	26.384	38.863	22.415
Python + C++	5	0.919	1.434	19.103	18.192	20.055	26.165	38.157	19.194
Python + C++ + Java	1	1.469	1.663	11.380	10.343	12.391	16.802	34.559	11.303
Python + C++ + Java	2	1.014	1.527	23.905	22.982	25.668	21.383	37.103	24.947
Python + C++ + Java	3	0.897	1.487	18.596	17.492	20.198	23.733	40.910	19.260
Python + C++ + Java	4	0.840	1.461	22.770	21.425	24.018	24.987	41.384	23.062
Python + C++ + Java	5	0.806	1.452	28.693	26.670	30.205	25.034	41.863	29.151

Table 2: Code Generation Model Evaluation Results

Model	Bleu Score	Sacre Bleu	ROUGE Score	ngram(w)	syntax	dataflow	code bleu	exact match
Conala (Python)	8.25	5.18	13.66	N/A	N/A	N/A	N/A	N/A
Python	20.73	22.64	47.35	12.09	14.28	19.58	14.90	3.56
Python + C	7.61	3.71	13.66	1.33	4.34	24.25	7.89	0
Python + C++	9.45	4.90	18.16	1.66	4.14	16.50	6.00	0
Python + C++ + Java	9.45	4.90	18.16	1.66	4.14	16.50	6.00	0

trend continued with the inclusion of C++ data, highlighting the model's resilience and adaptability to diverse paradigms.

However, incorporating Java data yielded mixed results. While some metrics improved, others declined slightly. This suggests potential challenges in assimilating Java's intricacies or reaching a saturation point in learning. Further investigation is needed to pinpoint the reason and optimize performance with Java and other languages.

4.3 Discussion of Evaluation Results

The Python model demonstrated the effectiveness of NLT transfer learning for code generation. Adding compiled languages like C and C++ did not significantly improve overall performance, suggesting limitations in the evaluation data or the need for further refinement. Notably, adding Java offered no improvements on any metrics.

The Code Bleu scores revealed a trade-off between syntactic similarity and functional accuracy. The Python model achieved high syntactic similarity, while the Python+C model performed better in terms of dataflow and functionality. This highlights the need to consider both aspects when evaluating code generation models.

4.4 Discussion of Results Discrepancies

The discrepancies between training and evaluator results highlight the need for further research into the model architecture and NLT transfer learning's generalizability. Potential reasons for these discrepancies include:

- **Data Bias:** The training data might be biased towards specific coding styles or domains, leading to overfitting and underperformance on the more diverse evaluation data. This bias may have also occurred as a result of subsequent fine tunings of languages.

- **Evaluation Metrics:** Training metrics focused primarily on syntactic similarity, while evaluator metrics might have placed greater

emphasis on functional correctness, code structure, or dataflow accuracy.

- **Compiled Language Challenges:** Adding compiled languages did not significantly improve performance, suggesting potential issues with the model's ability to handle their complexities or biases in the evaluation data.

- **Model Saturation:** Incorporating Java and data diversity as a whole led to a decline in performance on some metrics, indicating a potential saturation point.

5 Conclusion

5.1 Overview

This research has established a promising foundation for using NLT transfer learning to generate code across various languages. While the results highlight the effectiveness of the approach for interpreted languages like Python, incorporating compiled languages like C and C++ presented some challenges. This suggests the need for further refinement and investigation to achieve consistent and optimal performance across diverse programming paradigms.

5.2 Limitations and Future Work

Future research should focus on addressing identified limitations and enhancing the model's robustness and generalizability. Expanding the training data with larger and more diverse datasets, including more compiled languages and coding styles, is crucial. Additionally, optimizing the model architecture for specific languages like C++ and developing advanced evaluation metrics that consider both syntactic and functional aspects are essential. Exploring various code generation tasks and conducting additional studies would provide valuable insights for further improvement.

By addressing these crucial areas, we can significantly enhance the model's robustness, generalizability, and performance for universal application across all tasks and languages. With improved robustness and generalizability, code generation technologies can enable faster, more efficient, and more accurate code creation across various programming languages, leading to increased programmer productivity and opening new avenues for software development automation. This research continues the discussion for a future where code generation becomes a powerful tool, transforming the way software is developed and utilized.

References

Z. Alomari, O. El Halimi, K. Sivaprasad, and C. Pandit. 2015. Comparative studies of six programming languages. In *arXiv preprint arXiv:1504.00693*.

U. Alon, S. Brody, O. Levy, and E. Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *arXiv preprint arXiv:1808.01400*.

S. Chakraborty, M. Allamanis, and B. Ray. 2020. Tree2tree neural translation model for learning source code changes. In *arXiv preprint arXiv:1810.00314*.

A. Jha and C. K. Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *arXiv preprint arXiv:2206.00052*.

M. Junczys-Dowmunt, R. Grundkiewicz, T. Dwojak, H. Hoang, K. Heafield, T. Neckermann, F. Seide, U. Germann, A. F. Aji, N. Bogoychev, A. F. T. Martins, and A. Birch. 2018. Marian: Fast neural machine translation in c++. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 116–121.

D. KC and C. T. Morrison. 2023. Neural machine translation for code generation. In *arXiv preprint arXiv:2305.13504*.

T. H. M. Le, H. Chen, and M. A. Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. volume 53, pages 1–38.

Y. Liu, C. Tantithamthavorn, Y. Liu, and L. Li. 2023. On the reliability and explainability of language models for program generation. In *arXiv preprint arXiv:2302.09587*.

S. Norouzi, K. Tang, and Y. Cao. 2021. Code generation from natural language with less prior and more monolingual data. In *arXiv preprint arXiv:2101.00259*.

S. Sharma, N. Anand, and K. G. V. Kiran. 2023. Stochastic code generation. In *arXiv preprint arXiv:2304.08243*.

A. S. Soliman, M. M. Hadhoud, and S. I. Shaheen. 2022. Mariancg: a code generation transformer model inspired by machine translation. volume 69, page 104.

Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang. 2019. Treegen: A tree-based transformer architecture for code generation. In *arXiv preprint arXiv:1911.09983*.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2023. Attention is all you need. ArXiv preprint arXiv:1706.03762v7.

S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, and B. Xiang. 2022. Recode: Robustness evaluation of code generation models. In *arXiv preprint arXiv:2212.10264*.

P. Yin and G. Neubig. 2017. A syntactic neural model for general-purpose code generation. ArXiv preprint arXiv:1704.01696.

L. Zhang, C. Cao, Z. Wang, and P. Liu. 2023. Which features are learned by codebert: An empirical study of the bert-based source code representation learning. ArXiv preprint arXiv:2301.08427.