NCCS Research Journal, Volume 3, No. 1 October 2024, ISSN NO: 2822-1605

Received Date: June 2024 Revised: August 2024 Accepted: October 2024

Comparative Analysis of Transformer and CodeBERT for Program Translation

Bikash Balami

Assistant Professor

Central Department of Computer Science and Information Technology

Tribhuvan University

bikash@cdcsit.edu.np

Joshana Shakya

Machine Learning Engineer, Fusemachines

joshanashakya@gmail.com

Abstract

Program translation refers to the technical process of automatically converting the source code of a computer program written in one programming language into an equivalent program in another. This study compares the transformer model and the CodeBERT-based encoder-decoder model on the program translation task. Specifically, it trains the 6 and 12-layer models for 50 and 100 epochs to translate programs written in Java to Python and Python to Java. The models were trained with 3133 sets of Java Python parallel programs. Among different layered models, the transformer model with 6 layers trained for 50 epochs to translate from Java to Python achieved the highest BLEU and CodeBLEU scores, with values of 0.28 and 0.28, respectively. Similarly, the transformer model with 6 layers trained for 100 epochs to translate from Python to Java received the highest BLEU and CodeBLEU scores of 0.39 and 0.40, respectively. These results show that the transformer models perform better than the CodeBERT models. Also, the BLEU and CodeBLEU scores of the Java to Python and Python to Java translation models are different.

Keywords: *Program Translation, Transformer, Code Bidirectional Encoder Representations from Transformers, Bilingual Evaluation Understudy (BLEU), Code Bilingual Evaluation Understudy (CodeBLEU)*

Comparative Analysis of Transformer and CodeBERT for Program Translation

Software applications are computer programs that may become obsolete over time due to a variety of factors, including hardware platform updates, skills shortages in the original programming language in which the application was written, and a lack of software support from the language compiler vendors. As a result, software developers are often required to review software applications implemented in one programming language to a more recent and efficient language. Such reimplementation of any software needs knowledge of both programming languages: one that was used to develop the software and the other that will be used to rewrite the software.

Also, reimplementation is an expensive and time-consuming procedure. A bank in Australia, for example, spent $750 million in 5 years to migrate its core COBOL platform to Java (Lachaux et al., 2020). To reduce the risk and cost associated with code migration, developers often apply the simplest form of software re-engineering approach called program translation. Program translation is the technical process of automatically translating the source code of a computer program written in one language into an equivalent program in another language (Ahmad et al., 2023). Unlike traditional compilers, which translate a program written in a high-level programming language to a lower-level machine code (Java → Bytecode), the program translation system, also called a transcompiler, focuses on translation between high-level programming languages (Zhu et al., 2022).

Traditionally, program translation is performed in a rule-based manner, which involves parsing the input source code, constructing an abstract syntax tree (AST), transforming the AST, and finally generating source code in the target programming language. Given the dataset, the program written in one language can be translated to a different language without any programmatic intervention by employing a modern machine translation approach like neural machine translation (NMT). NMT is a machine learning approach to automate translation by utilizing neural networks.

As NMT was the recurrent neural network-based encoder-decoder model, this model has issues with long-range dependencies and non-parallelization within training examples. To deal with these issues, a novel transformer model was presented that achieved state-of-the-art on the WMT-14 English-to-German and English-to-French translation tasks and required significantly fewer

calculations and less time to train (Vaswani et al., 2017). The transformer-based NMT model can be trained by initializing the model weights to random values. Alternatively, the weights can be initialized by copying them from a previously trained model. This approach is called warm-starting. In the case of the programming language, the encoder-only model, Code Bidirectional Encoder Representations from Transformers (CodeBERT), can be used to warm-start the encoder and decoder of the NMT model.

Problem Statement

As programming languages can be considered as natural languages (Aggarwal et al., 2015), program translation problems can also be viewed as natural language translation problems. Therefore, different natural language translation approaches, such as rule-based, statistical machine translation (SMT), or NMT methods, can be applied to program translation problems. The transformer-based NMT architecture and the pretrained models improve the translation quality. In the case of the pretrained models, the CodeBERT encoders can also be used on the decoder side of the encoder-decoder model to have better output representation. And to reduce the memory usage as well as to reduce the execution time, the weights of encoders can be shared with those of decoders. The study attempts to provide answers of the following questions, How does encoder decoder model can be trained using parallel program data set?, How to compare different translation models? Do the Java to Python and Python to Java translation models yield similar scores?

Objectives

- To train the transformer and CodeBERT models on the Java – Python parallel program dataset
- To translate the program written in Java to Python and vice versa using Transformer and CodeBERT
- To compare the performance of the transformer and CodeBERT using BLEU and CodeBLEU as evaluation metrics

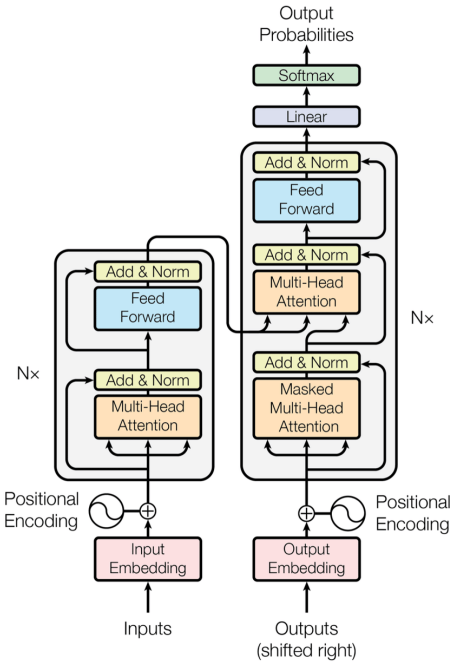
Theoretical Background

Transformer

A transformer is a deep learning model that utilizes the self-attention mechanism to solve sequence-to-sequence problems while resolving long-range dependencies. It is a type of artificial intelligence model that learns to understand and generate natural language text by analyzing patterns in large amount of text data. Transformers are a current state-of-the-art NLP model and are considered the evolution of the encoder-decoder architecture. This model avoids recurrence and trains the network in parallel to speed up the development of the model with a large number of parameters. Transformers are specially designed to comprehend context and meaning by analyzing the relationship between different elements, and they rely almost entirely on a mathematical technique called attention. The transformer architecture is shown in

Figure 1.

Transformer Model Architecture (Vaswani et al., 2017)



The model consists of two components: an encoder and a decoder. The encoder reads a sequence of symbol representations $x = (x_1, \ldots, x_n)$ as input and generates a sequence of continuous representations $z = (z_1, \ldots, z_n)$. Given z, the decoder produces a sequence of symbols $(y_1, \ldots, y_m)$ one element at a time (Vaswani et al., 2017).

The encoder block consists of *N* identical layers stacked on top of each other. Each layer contains two basic sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The decoder also consists of *N* identical layers stacked on top of each other. Each layer contains three sub-layers: a masked multi-head attention mechanism, a multi-head attention mechanism, and a position-wise fully connected feed-forward network.

Given the input tokens or the output tokens, the embedding sub-layer generates the vectors of dimensions d_{model} using learned embeddings . The learned linear transformation sublayer projects the vector produced by the stack of decoders to a logits vector, and the softmax function converts the vector to predicted next-token probabilities. Positional encoding add information about the relative or absolute position to input embeddings, positional encoding of dimension d_{model} is computed using sine and cosine functions of different frequencies. An attention function uses a query and a set of key-value pairs to calculate an attention. The compatibility function of the query with the corresponding key determines the weight allocated to each value.

CodeBERT

Bidirectional Encoder Representations from Transformers (BERT) is a language representation model based on the transformer architecture. The two types of BERT models based on the model size are BERTBase and BERTLarge. BERTBase has 12 transformer layers, 768 hidden size, 12 attention heads, and 110M trainable parameters, whereas BERTLarge has 24 transformer layers, 1024 hidden size, 16 attention heads, and 340M trainable parameters

The BERT model is designed to pretrain deep bidirectional representation using two tasks: masked language modeling (MLM) and next sentence prediction (NSP). During training the model, 15% of the tokens are masked, and the correct tokens in the masked positions are predicted using the final hidden state. NSP is used to learn the link between sentence pairs. For NSP, when choosing sentence pair A and B, 50% of the time it is an arbitrary sentence in the corpus. To predict the correct label and compute loss, the output hidden state is used. The pretrained BERT model can be used to fine-tune the downstream natural language processing tasks (Devlin et al., 2019).

CodeBERT is a bimodal pretrained model based on the transformer architecture for programming languages (PL) and natural language (NL). It learns the semantics connection between Pl and NL and supports downstream NL-PL tasks like natural language code search, code documentation generation, and so on. CodeBERT uses the RoBERTa-base architecture with 125M

model parameters (Feng et al., 2020). The CodeBERT is trained on both bimodal data (natural language–code) and unimodal data (code) across six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go) with a hybrid objective function (Feng et al., 2020).

Methodology

Data Collection

A parallel dataset for Java-Python program translation was collected from AVATAR: A Parallel Corpus for Java-Python Program Translation (Ahmad et al., 2023). The dataset contains Java and Python solutions to the programming problems. These solutions were taken from programming contest sites such as Codeforces, Google Code Jam, and online platforms such as GeeksforGeeks, LeetCode, and Project Euler. Hence, 20,363 parallel programs were taken. However, due to resource limitations, the programs having lengths less than 5 and greater than 450 were discarded after cleaning and pretokenization, leading to size 3133. For this study, as mentioned earlier the programs having length less than 5 and greater than 450 were discarded, so after that 3133 parallel program tasks were used for training and testing the model. The details about data can be seen in Table 1.

Table 1

Dataset Description

Source	Java Python Program Counts
Codeforces	1726
GeeksforGeeks	1354
LeetCode	35
Project Euler	18
Total	3133

Out of 3133, 80% (2506) were used for training and the remaining 20% (627) were used as test samples. Sample of the dataset is presented in Figure 2 and Figure 3.

Figure 2

Program written in Java

```
import java.io.*;
class GfG {
static int sumOfTheSeries(int n){
return (n * (n + 1) / 2) *
(2 * n + 1) / 3;
}
public static void main (String[] args)
{
int n = 5;
System.out.println("Sum = "+
sumOfTheSeries(n));
}}
```

Figure 3

Program in Python of code of program written in Java in Figure 2

```
def sumOfTheSeries( n ):
return int((n * (n + 1) / 2) *
(2 * n + 1) / 3)
n = 5
print("Sum =", sumOfTheSeries(n))
```

Data Preprocessing

The dataset of Java and Python programs were processed through the preprocessing tasks to obtain data suitable to train the models. The tasks include data cleaning, pretokenization and tokenization. Unlike other programming languages, indentation is a crucial concept that should be followed when writing Python code. Moreover, Python does not allow mixing tabs and spaces for indentation. However, Python programs in Project Euler have IndentationError, which was fixed using autopep8. Following this, pyminifier was used to remove the docstrings, comments, and extraneous whitespaces present in each Python program, as well as to minimize indentation spaces. The pyminifier uses a single space to substitute multiple whitespaces or tabs used as an indentation in the program.

In pretokenization phase, each program was split into meaningful code tokens. For the transformer model, each Java program was tokenized using javalang. The javalang tokenizer generates a stream of Java tokens, each having position (line, column) and value information. It also removes code comments. Each Python program was tokenized using tokenize from the Python library. For CodeBERT, pretokenization of a Java program was done by splitting the program into tokens, detokenizing those tokens using javalang, and then binding tokens using a space character. Tokenization is the process of splitting a text into words, phrases, or other meaningful elements called tokens. In this step, each pretokenized program was split into smaller subunits using a subword tokenization approach called Byte Pair Encoding (BPE). A BPE has two parts: a token learner that generates a vocabulary from a raw training corpus and a token segmenter that tokenizes a raw program based on the vocabulary.

Neural Translation Model

Two neural translation models were built using an encoder and decoder model: one with a transformer architecture and the other with both the encoder and the decoder initialized with the public CodeBERT checkpoint.

Inference

To generate translations from a probability model, the Greedy 1-best search criterion was used. In greedy search, the probability at every time step is calculated and the token that gives the highest probability is selected to use as the next token in the sequence.

Data Postprocessing

The programs translated using the transformer were postprocessed by first removing BPE tokens and "<unk>" tokens. In the case of Java program, the program tokens were detokenized by simply reformatting using javalang. For a Python program, any text in capital or small letters matching "newline" and "new line" was replaced with the text "NEWLINE", "indent" with "INDENT", and "dedent" with "DEDENT". Following that, the program was detokenized by splitting it on "NEWLINE", replacing "INDENT" appearing at the beginning of each line with four spaces, and removing the texts "INDENT", "DEDENT", "NL", and "ENDMARKER". Finally, all the lines were joined with the "\n" character. In the resulting program, ". " and " ." were replaced with "." and minified using pyminifier.

Evaluation

The BLEU score and the CodeBLEU score were used to assess the transformer and CodeBERT models' performance. Both the BLEU and CodeBLEU scores ranges from 0 to 1, with 0 indicating a perfect mismatch and 1 indicating a perfect match. The models were evaluated under the hyper parameters described in Table 2.

Table 2

Hyperparameters Description

No. of layers	(6,6), (12,12)
No. of heads	12
Embedding size	768
FFN Hidden Dimension	3072
Activation	GELU
Dropout	0.1
Layer normalization epsilon	1e-12
Loss function	Cross Entropy Loss
Optimizer	Adam Optimizer
Batch size	16
Learning rate	2e-5
No. of epochs	50,100

Result Analysis

The experiment was conducted on different configurations of the transformer and the CodeBERT model, for varying numbers of epochs. The sample of output is shown in Figure 4 and Figure

Figure 4

Program written in Java 5

```
import java.io.*;

class GFG {
static long calculateSum(int n)
{
long sum = 0;
for (int row = 0; row < n; row++) {
sum = sum + (1 << row);
}
return sum;
}
public static void main(String[] args)
{
int n = 10;
System.out.println("Sum of all elements:"
+ calculateSum(n));
}}
```

Figure 5

Generated Python code of Figure 4

```
def calculateSum(n) :
sum = 0
for row in range(n):
sum = sum + (1 << row)
return sum
n = 10
print("Sum of all elements:", calculateSum(n))
```

The BLEU and CodeBLEU scores of the transformer and CodeBERT models obtained for 627 testing samples are shown in Table 3 (Java to Python) and Table 4 (Python to Java).

Table 3

BLEU, CodeBLEU, scores for Java to Python translation

Epoch	Layers	Model	BLEU	Code BLEU
50	6	Transformer	0.28	0.28
		CodeBERT	0.13	0.21
	12	Transformer	0.26	0.27
		CodeBERT	0.10	0.19
100	6	Transformer	0.27	0.27
		CodeBERT	0.15	0.22
	12	Transformer	0.25	0.27
		CodeBERT	0.11	0.17

Table 4

BLEU, CodeBLEU, scores for Python to Java translation

Epoch	Layers	Model	BLEU	Code BLEU
50	6	Transformer	0.39	0.40
		CodeBERT	0.27	0.35
	12	Transformer	0.37	0.38
		CodeBERT	0.26	0.33
100	6	Transformer	0.39	0.40
		CodeBERT	0.29	0.37
	12	Transformer	0.37	0.38
		CodeBERT	0.29	0.37

In order to choose the appropriate model for program translation, the study evaluates the BLEU and CodeBLEU scores of the CodeBERT and the transformer models. The study uses trained models to translate programs from the test dataset and compute BLEU and CodeBLEU scores. It determines whether the Python to Java program translation models have equivalent BLEU and .CodeBLEU scores to the Java to Python program translation models. The results demonstrated that the transformer model with 6 encoder and 6 decoder layers, trained for 50 epochs to translate from Java to Python, received the highest BLEU and CodeBLEU scores, with values of 0.28 and 0.28, respectively. Similarly, the transformer model, trained for 100 epochs to translate from Python to Java, achieved the highest BLEU (0.39) and CodeBLEU (0.40) scores.

Additionally, the results demonstrate that Python to Java translation models have higher BLEU and CodeBLEU scores than Java to Python translation models. In this study, the transformer models have performed better than the CodeBERT models in terms of BLEU and CodeBLEU scores.

Conclusion

The purpose of this study was to compare the transformer and the CodeBERT model on program translation tasks. The study used a 3133 Java Python parallel program dataset to translate the programs written in the source language to the target language. 80% (2506) of the data was used to train and 20% (672) of the data was used to test the transformer and the CodeBERT models.

Based on the BLEU and CodeBLEU scores of the models trained for different epochs, it can be concluded that the transformer models performed better than the CodeBERT models on the test dataset used in the study.

For the Java to Python program translation task, the transformer model with 6 encoder and 6 decoder layers trained for 50 epochs achieved the highest BLEU and Code BLEU scores, of 0.28 and 0.28, respectively.

Similarly, for the Python to Java program translation task, the transformer model with 6 encoder and 6 decoder layers trained for 100 epochs received the highest BLEU and CodeBLEU scores, with values of 0.39 and 0.40, respectively. Furthermore, the scores of Java-to-Python translation models differ from those of Python to Java translation models.

Future Enhancements

The study used the CodeBERT block on both the encoder and decoder side of the translation model with shared weights. It is possible to use an autoregressive model on the decoder side. Additionally, due to resource constraints, the experiment was run on a small set of data. It would have been good if all of the datasets were used to train the models.

References

Aggarwal, K., Salameh, M., & Hindle, A. (2015). Using machine translation for converting Python 2 to Python 3 code. *PeerJPreprints*.

Ahmad, W. U., Tushar, M. G., Chakraborty, S., & Chang, K.-W. (2023). AVATAR: A Parallel Corpus for Java-Python Program Translation. *arXiv:2108.11590v2*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805v2*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., . . . Zhou, M. (2020). CodeBERT: APre-Trained Model for Programming and Natural Languages. *arXiv:2002.08155v4*.

Lachaux, M.-A., Roziere, B., & Chanussot, L. (2020). Unsupervised Translation of Programming Languages. *arXiv:2006.03511v3*.

Vaswani, A., M.Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., N.Gomez, A., . . . Polosukhin, I. (2017). Attention is All you Need. *Neural Information Processing*.

Zhu, M., Suresh, K., & K.Reddy, C. (2022). Multilingual Code Snippets Training for Program Translation. *Association for the Advancement of Artificial Intelligence*.

Acknowledgments

We are greatly thankful to Asst. Prof. Sarbin Sayami, Head of the Central Department of Computer Science & Information Technology (CDCSIT), Kirtipur for providing valuable suggestions, support, and inspiration to complete this task.

We are thankful to Prof. Dr. Shashidhar Ram Joshi, Prof. Dr. Subarna Shakya, Prof. Dr. Bal Krishna Bal and Assoc. Prof. Nawaraj Paudel for their kind suggestions and inspiration during this work.