Tribhuvan University
Institute of Science and Technology

"Comparative Analysis of Deep Learning Models (Transformer and CodeBERT) for Program Translation"

Dissertation

Submitted To:

Central Department of Computer Science and Information Technology

Tribhuvan University,

Kirtipur, Kathmandu

Nepal

In partial fulfillment of the requirements for the Degree of Master of Science in Computer Science and Information Technology

Submitted By:

Joshana Shakya

July, 2022

Supervisor

Asst. Prof. Bikash Balami

CDCSIT, TU

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Student's Declaration

I hereby declare that I am the only author of this work and that no sources other than the listed here have been used in this work.

. . . . . .

Joshana Shakya

Date: July, 2022

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Supervisor's Recommendation

I hereby recommend that this dissertation prepared under my supervision by **Ms. Joshana Shakya** entitled **"Comparative Analysis of Deep Learning Models (Transformer and CodeBERT) for Program Translation"** in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science and Information Technology be processed for the evaluation.

.....

Asst. Prof. Bikash Balami

Central Department of Computer Science and Information Technology

Tribhuvan University, Nepal

Date: July, 2022

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

LETTER OF APPROVAL

We certify that we have read this dissertation and in our opinion it is satisfactory in the scope and quality as a dissertation in the partial fulfillment for the requirement of Master's Degree in Computer Science and Information Technology.

Evaluation Committee

.....

Asst. Prof. Sarbin Sayami

Head of Department

Central Department of Computer Science and Information Technology

Tribhuvan University, Nepal

.....

Asst. Prof. Bikash Balami

(Supervisor)

Central Department of Computer Science and Information Technology

Tribhuvan University, Nepal

.....

(External Examiner)

Data: July, 2022

.....

(Internal Examiner)

ACKNOWLEDGEMENT

First of all, I would like to express my deepest sense of gratitude and sincere thanks to my respected supervisor, Mr. Bikash Balami, Assistant Professor, Central Department of Computer Science and Information Technology, Tribhuvan University for his valuable guidance, encouragement, and help for completing this research work. Without his involvement in every step, this work would have never been completed.

I would like to express my sincere thanks to the respected Head of Department Asst. Prof. Sarbin Sayami, Central Department of Computer Science and Information Technology, for whole hearted support. I am also grateful to all Professors and Lecturers of the Department for their constant support and guidance.

In the end, I would like to express my sincere thanks to my family, friends, and others who helped me directly or indirectly while writing this dissertation.

Joshana Shakya

July, 2022

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. Deep learning models like the transformer and Code Bidirectional Encoder Representations from Transformers (CodeBERT) models can be trained to perform such program translation. 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.

A total of 3133 Java-Python parallel programs were collected, and then the models were trained using the preprocessed training data. To compare the models, the Bilingual Evaluation Understudy (BLEU) and CodeBLEU scores were calculated on the test dataset. 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 0.2812 and 0.2802, 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 0.3891 and 0.4018, 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: *Machine Translation, Program Translation, Transformer, Code Bidirectional Encoder Representations from Transformers (CodeBERT), Bilingual Evaluation Understudy (BLEU), Code Bilingual Evaluation Understudy (CodeBLEU)*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
BERT	Bidirectional Encoder Representations from Transformers
BLEU	Bilingual Evaluation Understudy Score
BPE	Byte Pair Encoding
CodeBERT	Code Bidirectional Encoder Representations from Transformers
DAE	Denoising Auto-Encoding
DM	Dataflow Match
GELU	Gaussian Error Linear Unit
GPT	Generative Pretrained Transformer
GPU	Graphical Processing Unit
MLM	Masked Language Modeling
NL	Natural Language
NMT	Neural Machine Translation
NSP	Next Sentence Prediction
PL	Programming Language
PLBART	Program and Language Bidirectional and Auto-Regressive Transformers
ReLU	Rectified Linear Unit
RoBERTa	Robustly Optimized BERT Approach
RTD	Replaced Token Detection
SM	Syntactic AST Match
SMT	Statistical Machine Translation
WNM	Weighted N-gram Match

CHAPTER 1

INTRODUCTION

1.1 Introduction

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 [1]. As a result, software developers are often required to rewrite 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 [2]. To reduce the risk and cost associated with code migration, developers often apply the simplest form of software re-engineering approach called program translation [1].

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 [3]. Unlike traditional compilers, which translate a program written in a high-level programming language to a lower-level machine code (Java $\rightarrow$ Bytecode), the program translation system, also called a transcompiler, focuses on translation between high-level programming languages [4]. The quality of the transcompiler decides whether or not the translated code need manual editing to function properly.

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 [5]. This process is illustrated in Figure 1.1. Similarly, provided 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).

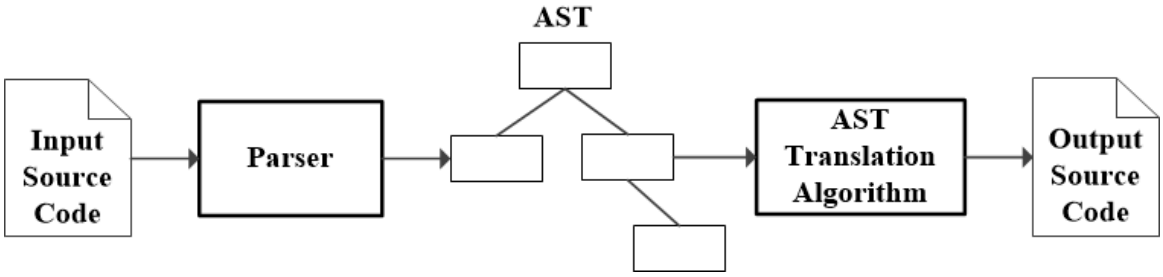


Figure 1.1: Program translation using AST [5]

NMT is a machine learning approach to automate translation by utilizing neural networks. Through training on the datasets, the NMT model captures the source and target connections and learns to predict and increase the probability of correct translations [6]. A few years back, the most popular architecture for NMT was the recurrent neural network based encoder-decoder model. But, 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 the 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 [7]. 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 [8]. 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.

1.2 Problem Statement

As programming languages can be considered as natural languages [9], 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 rule-based program translation method is inefficient and time-consuming [10]. Also, the SMT technology delivers lower quality translation and is time-consuming in comparison to NMT [11]. 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, the weights of encoders can be shared with those of decoders.

1.3 Research Questions

The study attempts to provide answers of the following questions:

i. How does the encoder-decoder model initialized with the public CodeBERT checkpoint perform on the program translation task in comparison with a transformer model, as measured with the Bilingual Evaluation Understudy Score (BLEU) and CodeBLEU scores?
ii. What are the BLEU and CodeBLEU scores of the Java to Python and Python to Java translation models on the test dataset?
iii. Do the Java to Python and Python to Java translation models yield similar scores?

1.4 Objectives

The general and specific objectives of the study are:

General Objective: To compare the deep learning models, transformer and CodeBERT, on the translation task.

Specific Objectives

i. To train the transformer and CodeBERT models on the Java-Python parallel program dataset.
ii. To compare the performance of the trained transformer and CodeBERT models using BLEU and CodeBLEU evaluation metrics.
iii. To translate programs written in Java to Python and vice-versa using the trained transformer and CodeBERT models.

1.5 Dissertation Organization

This dissertation is organized as follows:

Chapter 1 consists of the introduction, problem statement, research questions, and objectives of this dissertation.

Chapter 2 discusses the theoretical background that provides theoretical details of the components used in this dissertation.

Chapter 3 includes the literature review of the existing works related to the transformer, CodeBERT, and translation.

Chapter 4 describes the methodology used to compare the transformer and CodeBERT

models.

Chapter 5 explains the implementation tools, test environment, and hyperparameters used.

Chapter 6 presents the results and analysis of the results.

Chapter 7 includes the dissertation conclusion and limitations. It also provides future recommendations.

CHAPTER 2

THEORETICAL BACKGROUND

2.1 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. This model avoids recurrence and trains the network in parallel to speed up the development of the model with a large number of parameters. The transformer architecture is shown in Figure 2.1.

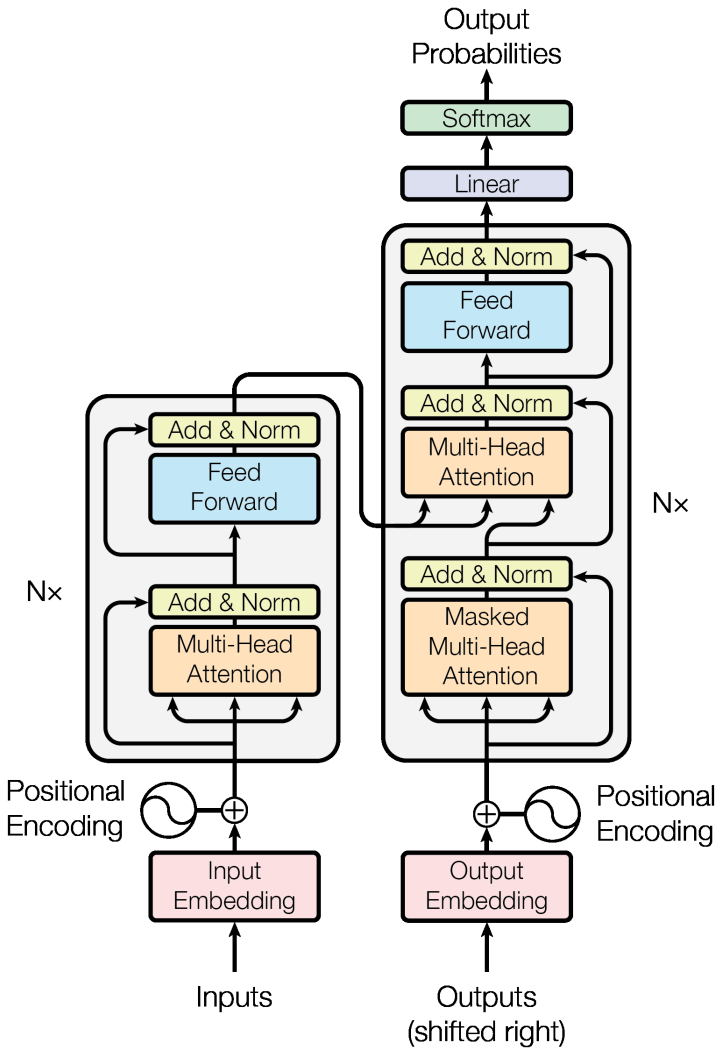


Figure 2.1: Transformer model architecture [7]

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.

i. Encoder

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, as shown in Figure 2.1(left). The residual connection surrounds each of the two sub-layers, $SubLayer(x)$, to forward the unprocessed input x of the sub-layer to a layer normalization function. Thus, each sub-layer produces an output:

$$LayerNorm(x + SubLayer(x)) \tag{2.1}$$

The residual connections, the output of all the sub-layers in the model, including the embedding layers, has a constant dimension d_{model} [7], [12].

ii. Decoder

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. The structure of the single decoder layer is shown in Figure 2.1(right). The masked multi-head attention mechanism prevents from looking into the future positions, and the multi-head attention mechanism works over the output of the encoder stack. Similar to the encoder, the residual connection surrounds each of the three sub-layers, $Sublayer(x)$ [7], [12].

2.1.1 Embeddings and Softmax

Given the input tokens or the output tokens, the embedding sub-layer generates the vectors of dimensions d_{model} using learned embeddings [7]. The learned linear transformation sub-layer projects the vector produced by the stack of decoders to a logits vector, and the softmax function converts the logit vector to predicted next-token probabilities [13]. The two embedding layers and linear transformation have the same weight matrix, but in the embedding layers those weights are multiplied by $\sqrt{d_{model}}$ [7].

2.1.2 Positional Encoding

To 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:

$$PE_{(pos,\ 2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.2}$$

$$PE_{(pos,\ 2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.3}$$

where pos is the position and i is the dimension [7].

2.1.3 Attention

An attention function uses a query and a set of key-value pairs to calculate an attention. The attention is a weighted sum of the values. The compatibility function of the query with the corresponding key determines the weight allocated to each value.

2.1.3.1 Scaled Dot-Product Attention

Figure 2.2 shows a scaled dot-product attention.

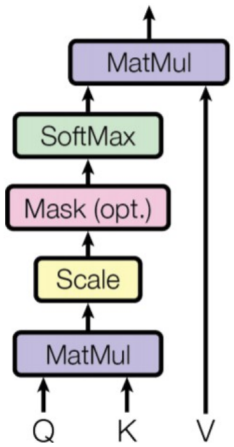


Figure 2.2: Scaled dot-product attention [7]

The scaled dot-product attention takes queries of dimension d_k, keys of dimension d_k, and values of dimension d_v, as inputs. It computes the attention function on a set of queries, keys, and values packed together into matrices Q, K, and V respectively as [7]:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.4}$$

2.1.3.2 Multi-Head Attention

The multi-head attention mechanism linearly projects the queries, keys, and values to d_k, d_k, and d_v dimensions, respectively, h times with different learned projections. The h heads are run in parallel to obtain d_v-dimensional output values. The outputs of h heads are then concatenated as:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{2.5}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.6}$$

where parameter matrices, $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$ are the projections [7]. The structure of multi-head attention is depicted in Figure 2.3.

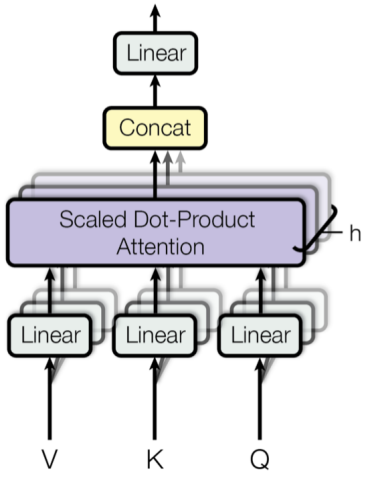


Figure 2.3: Multi-head attention [7]

The transformer model employs multi-head attention as:

i. Encoder-decoder attention

   Here, the attention function uses queries from the previous decoder layer and the memory keys and values from the output of the encoder.

ii. Encoder self-attention

   The encoder self-attention uses keys, values, and queries from the output of the previous layer of the encoder.

iii. Masked decoder self-attention

   It permits self-attention to focus on earlier positions in the output sequence by masking future positions (setting them to -inf) prior to the softmax step in the calculation [13].

2.1.4 Position-wise Feed-Forward Network

The feed-forward network contains two layers and applies a Rectified Linear Unit (ReLU) activation function in between.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{2.7}$$

2.1.5 Post-layer Normalization

Post-layer normalization comes after each attention sub-layer and each feed-forward sub-layer. It has an add function and a layer normalization process. The add function processes the residual connections from the input of the sub-layer to ensure that the critical information is not lost. The layer normalization function is defined as [12]:

$$LayerNorm(v) = \gamma \frac{v - \mu}{\sigma} + \beta \tag{2.8}$$

where $v = x + SubLayer(x)$,

$\mu = \frac{1}{d} \sum_{k=1}^{d} v_k$ is the mean of v of dimenstion d,

$\sigma = \sqrt{\frac{1}{d} \sum_{k=1}^{d} (v_k - \mu)^2}$ is the standard deviation of v of dimention d,

γ is a scaling parameter,

and β is a bias vector.

2.2 BERT

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 general architecture of BERT is shown in Figure 2.4.

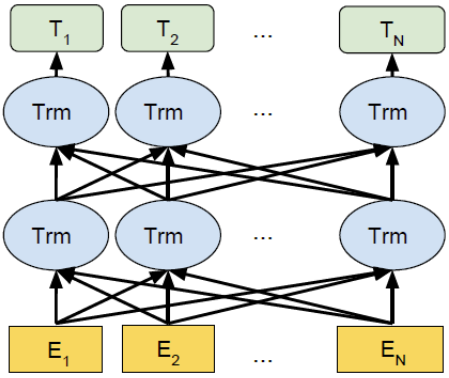


Figure 2.4: BERT architecture [14]

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 B is the actual next sentence after A, and 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 [14].

2.2.1 RoBERTa

Robustly optimized BERT approach (RoBERTa) is a replication of BERT with the following useful modifications [15]:

i. Dynamic masking

   BERT employs static masking during preprocessing, whereas RoBERTa employs dynamic masking wherein different parts of the sentences are masked in different epochs.

ii. No NSP task

   The NSP task is not used to train the RoBERTa model.

iii. More data points

   RoBERTa was trained on more datasets which include Common Crawl-News, Open WebText, etc., in addition to the Toronto BookCorpus and English Wikipedia datasets with which BERT was trained.

iv. Large batch size

   BERT used a batch size of 256 with 1 million steps, whereas RoBERTa used a batch size of 8,000 with 300,000 steps.

2.2.2 CodeBERT

CodeBERT is a bimodal pretrained model based on the transformer architecture for programming languages (PL) and natural language (NL). It learns the semantic 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 [16], [17].

2.2.2.1 Pretraining CodeBERT

In the pretraining phase, the input is set as:

$$[CLS], w_1, w_2, \ldots, w_n, [SEP], c_1, c_2, \ldots, c_m, [EOS]$$

where $w_1, w_2, \ldots, w_n$ is a natural language text,

$c_1, c_2, \ldots, c_m$ is a code,

$[CLS]$ is a special token whose final hidden representation works as the aggregated sequence representation,

$[SEP]$ is a separator token,

and $[EOS]$ is End of Sequence token.

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 which includes [16]:

i. Masked Language Modeling

Given a datapoint, $x = w, c$, where w is a NL word sequence and c is a PL token sequence, 15% of the tokens from x are replaced with [MASK] tokens at random NL positions m^w and PL positions m^c.

$$m_i^w \sim unif\{1, |w|\}\ for\ i = 1\ to\ |w| \tag{2.9}$$

$$m_i^c \sim unif\{1, |c|\}\ for\ i = 1\ to\ |c| \tag{2.10}$$

$$w^{masked} = REPLACE(w, m^w, [MASK]) \tag{2.11}$$

$$c^{masked} = REPLACE(c, m^c, [MASK]) \tag{2.12}$$

$$x = w + c \tag{2.13}$$

The MLM objective is formulated as follows:

$$L_{MLM}(\theta) = \sum_{i \in m^w \cup m^c} -logp^{D_1}(x_i | w^{masked}, c^{masked}) \tag{2.14}$$

where p^{D_1} predicts a token from a large vocabulary.

ii. Replace Token Detection

The replaced token detection (RTD) objective has an NL generator p_w^G and a PL generator p_c^G to produce alternatives for the randomly masked positions.

$$\hat{w}_i \sim p^{G_w}(w_i | w^{masked})\ for\ i \in m^w \tag{2.15}$$

$$\hat{c}_i \sim p^{G_c}(c_i|c^{masked}) \; for \; i \in m^c \tag{2.16}$$

$$w^{corrupt} = REPLACE(w, m^w, \hat{w}) \tag{2.17}$$

$$c^{corrupt} = REPLACE(c, m^c, \hat{c}) \tag{2.18}$$

$$x^{corrupt} = w^{corrupt} + c^{corrupt} \tag{2.19}$$

The discriminator, which is a binary classification problem, is trained to find out whether a word is the original or not. The RTD objective is formulated as:

$$\begin{aligned} L_{RTD}(\theta) = \sum_{i=1}^{|w|+|c|} \big(&\delta(i) log p^{D_2}(x^{corrupt}, i) + \\ &(1-\delta(i))(1 - log p^{D_2}(x^{corrupt}, i))\big) \end{aligned} \tag{2.20}$$

$$\delta(i) = \begin{cases} 1, & if \; x_i^{corrupt} = x_i \\ 0, & otherwise \end{cases} \tag{2.21}$$

where p^{D_2} is the probability of the i^{th} word being original,

and $\delta(i)$ is an indicator function.

The combined MLM and RTD loss function is:

$$min_\theta \; L_{MLM}(\theta) + L_{RTD}(\theta) \tag{2.22}$$

2.3 Gaussian Error Linear Unit

Gaussian Error Linear Unit (GELU) is an activation function with the following equation [18]:

$$GELU(x) = 0.5x \left(1 + tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right) \tag{2.23}$$

2.4 Cross Entropy Loss

Cross entropy loss is a function to compute the differences between the estimated probability and the actual probability. It is defined as [19]:

$$H(\hat{y}, y) = -\sum_{j=1}^{m} y_j log(\hat{y}_j) \tag{2.24}$$

2.5 Adam Optimization Algorithm

The Adam optimization algorithm updates the parameters of the model as follows [20]:

Algorithm 1 Adam Optimization Algorithm

Initialize $V_{dW} = 0, S_{dW} = 0, V_{db} = 0, S_{db} = 0$

On iteration t:

  Compute dW, db using current mini-batch

  $V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$

  $V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$

  $S_{dW} = \beta_1 S_{dW} + (1 - \beta_2) dW^2$

  $S_{db} = \beta_1 S_{db} + (1 - \beta_2) db^2$

  $V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t}$

  $V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$

  $S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}$

  $S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$

  $W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$

  $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$

The default values of the hyperparameters are β_1 = 0.9, β_2 = 0.999, and ϵ = 10^{-8}.

2.6 Evaluation Metrics

2.6.1 BLEU

The BLEU algorithm calculates the precision score of a candidate machine translation against a reference translation. The reference translation is a sample translation. The algorithm finds all of n-gram matches and determines the strength of the match with the precision score. The precision score is the fraction of n-grams in the translation that also appear in the reference. Let k be the maximum n-gram considered to evaluate the score of translation. The n-gram precision score is:

$$p_n = \frac{\#\ matched\ n\text{-}grams}{\#\ n\text{-}grams\ in\ candidate\ translation} \tag{2.25}$$

Let $w_n = \frac{1}{2n}$ be a geometric weighting for the precision of the n'th gram. The brevity penalty is:

$$\beta = exp^{min\left(0, 1 - \frac{len_{ref}}{len_{MT}}\right)} \tag{2.26}$$

where len_{ref} is the length of the reference translation and len_{MT} is the length of the machine translation.

The BLEU score is defined as [19]:

$$BLEU = \beta \prod_{i=1}^{k} p_n^{w_n} \tag{2.27}$$

2.6.2 CodeBLEU

CodeBLEU is defined as the weighted combination of the four components as follows [21]:

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weight} + \gamma \cdot Match_{ast} + \delta \cdot Match_{df} \tag{2.28}$$

where $BLEU$ is the standard BLEU score,

$BLEU_{weight}$ is the weighted n-gram match (WNM),

$Match_{ast}$ is the syntactic AST match (SM),

$Match_{df}$ is the semantic dataflow match (DM),

and α, β, γ, and δ are hyperparameters.

CHAPTER 3

LITERATURE REVIEW

Nguyen *et al.* [22] have studied SMT models to migrate source code written in one programming language into another. They used Phrasal, a phrase-based SMT model, to perform method-level translation from Java to C#. For training Phrasal, they treated each method and each code token from two open-source projects, db4o and Lucene, as a sentence and a word, respectively. The results show that SMT is a viable approach for performing source code migration.

To adapt codes written in Python 2 to Python 3, Aggarwal *et al.* [9] employed a SMT approach. For this purpose, the authors used a tool called 2to3 to convert Python 2 projects—Django and Natural Language Toolkit—to Python 3; they used Giza++ for aligning codes and Moses for constructing the translation model. The high BLEU scores achieved in their experiment demonstrate that traditional SMT models can be used to translate programs written in Python 2 to Python 3 with some postprocessing. And according to the authors, programming languages can be viewed as natural languages.

Roziere *et al.* [2] have proposed an unsupervised neural transpiler called TransCoder. The TransCoder is an encoder-decoder model based on the transformer architecture. To build this model, they constructed monolingual datasets using the functions extracted from C++, Java, and Python open source projects available on GitHub. The authors trained the model using the concept of unsupervised neural machine translation. First, they performed cross-lingual programming language model pretraining with a masked language modeling objective. Second, they trained the model with a Denoising Auto-Encoding (DAE) objective to encode and decode data sequences. Finally, they used a back-translation approach to increase the translation quality. The authors built the TransCoder model with a transformer having 6 layers, 8 attention heads, and 1024 dimensions, and used beam search decoding for inference. They experimented with the model by translating the source codes from GeeksforGeeks and evaluating it using computational accuracy, reference match, and BLEU score. With TransCoder, the authors demonstrated that the model can be used for any programming language without any specific knowledge.

In [23], Hassan *et al.* have built a transformer-based code converter to translate Java codes to Swift and Swift codes to Java. Due to the unavailability of the Java-Swift parallel dataset, the authors generated synthetic data, which consists of random independent codes with arbitrary variable names and literals. They trained the transformer model using Open-NMT. The model had a token accuracy ranging from 70% to 90%.

Rothe *et al.* [24] have experimented with BERT, Generative Pretrained Transformer (GPT) 2, and RoBERTa models. They used the pretrained version of these models to initialize a number of sequence-to-sequence models. The different combination of models were then tested on several tasks such as summarization, translation, etc. The evaluation of the different combinations of the models shows that BERT2RND, which is BERT with a randomly initialized decoder, and BERTShare, were the best performing models in terms of translation.

Ahmad *et al.* [3] have presented a Java-Python parallel dataset constructed from the solutions of 8,475 programming problems. The authors trained the NMT models from scratch as well as fine-tuned a few pretrained models, viz. CodeGPT, CodeBERT, GraphCodeBERT, and Program and Language Bidirectional and Auto-Regressive Transformers (PLBART). The experimental results show that the translation models did perform well in terms of lexical matching. However, the models were not able to produce syntactically correct codes with significant data-flow matching. Based on the results of the performance evaluation, PLBART was the best performing model with a 67.1 BLEU score and a 43.3 CodeBLEU score. In terms of BLEU score, the transformer and CodeBERT models did well.

Tauda *et al.* [25] have designed a translator for server-side and client-side programming languages which is based on the Long Short-Term Memory model. The authors trained the model twice using the 102 pairs of data types and decorators of Typescript and the Kotlin language. The model was built to translate applications written using the NestJS framework to client-based Android applications.

In [26], Feng and Chiba have introduced an approach for improving the quality of programming language translation. In this approach, the authors interpreted source code in terms of the syntax tree, as the syntax tree contains a lot of information about the structure of

code. The authors compared this syntax tree representation of code to the text representation of code in program translation from Java to Python and vice-versa. They trained the transformer-based translation model using an unsupervised neural machine translation approach. The translation results show that the translations have a very low BLEU score. However, the model trained with their syntax tree-based approach was able to generate important structures like conditional statements and loops when compared to the model trained with the text representation of code.

In [27], Shah *et al.* have attempted to translate natural language to the Python programming language. They trained the transformer model using the Django dataset. The translation model was able to learn the Python syntax, but it had difficulty understanding variable names.

To fix a defective Java program, Mashhadi and Hemmati [28] suggested an automated CodeBERT-based approach. The authors used an encoder-decoder model to construct the program repair model. On the encoder part, they used CodeBERT, and on the decoder part, the authors used a transformer decoder. The accuracy of the model shows that this approach is a feasible solution to fix bugs.

To study whether the representations yielded from the pretrained models contain the attributes of the source code, Karmakar and Robbes [29] have introduced four tasks. These tasks determine whether the code representations obtained from the pretrained models contain surface-level, syntactic, structural, and semantic information. The tasks include checking if the model can generate a valid token, predicting code complexity, predicting code length, and detecting invalid types. For evaluation, the authors trained a classifier which takes as input the hidden layer of a pretrained model. The probe analyses showed that CodeBERT delivered promising results in terms of syntactic and semantic understanding.

In [4], Zhu *et al.* have presented a parallel dataset containing data from 7 different programming languages. The authors used a transformer model with 12 encoder layers, 6 decoder layers, 768 dimensions, and 12 attention heads. They initialized the model with the pretrained weights. To understand the similarities between different languages, the authors first trained the model with the DAE objective on a snippets dataset. Next, the model was

pretrained for multilingual snippet translation to improve the quality of program translation. The authors used code snippets for two reasons: snippets are shorter in length than programs, and snippets help to learn long-distance dependencies. The BLEU and CodeBLEU scores show that DAE and multilingual snippet pretraining improved the translation performance of the model.

CHAPTER 4

METHODOLOGY

The purpose of the study was to assess the performance of deep learning models for program translation tasks. For this, the models were trained to transform the programs written in the source language into the programs in the target language. The different steps involved in this development and the evaluation of the models are illustrated by the diagram in Figure 4.1.

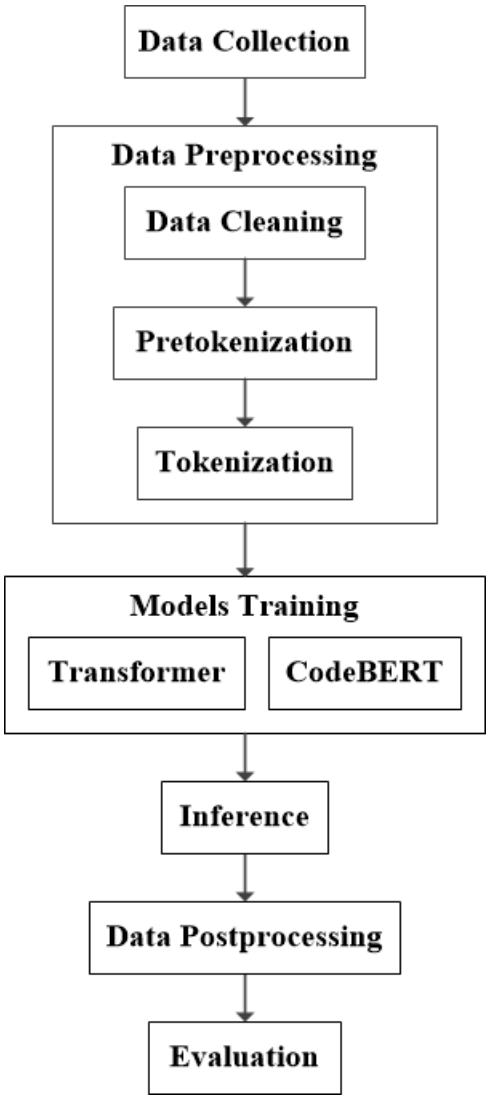


Figure 4.1: Process flow diagram outlining methodological steps

4.1 Data Collection

A parallel dataset for Java-Python program translation was collected from AVATAR: A Parallel Corpus for Java-Python Program Translation [3]. The dataset contains Java and

Python solutions to the programming problems. These solutions were taken from programming contest sites such as Codeforces, AtCoder, Google Code Jam, and online platforms such as GeeksforGeeks, LeetCode, and Project Euler. For the study, the dataset from AtCoder was not used, as there was no clear separation of equivalent Java and Python programs. 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. The details of the dataset are shown in the Table 4.1.

Table 4.1: Dataset description

Source	**Java-Python programs count**	**Java-Python programs count ($5 \leq$ length ≤ 450)**
Codeforces	11205	1726
Code Jam	3546	0
GeeksforGeeks	5289	1354
LeetCode	122	35
Project Euler	201	18
Total	**20363**	**3133**

Of 3133 parallel programs, 80% (2506) were used for training and the remaining 20% (627) were used as test samples.

4.2 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.

4.2.1 Data Cleaning

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.

4.2.2 Pretokenization

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. From the generated Python tokens, tokens of type "COMMENT" and type "NL" following the type "COMMENT" were removed. As newlines and spaces define the structure of the Python code, all tokens of type "NEWLINE" were replaced with the text "NEWLINE", type "NL" with the text "NL", type "INDENT" with the text "INDENT", type "DEDENT" with the text "DEDENT", and type "ENDMARKER" with the text "ENDMARKER". All the tokens were then joined together by a space character.

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. In the case of a Python program, tokens of type "COMMENT" and "NL" following the type "COMMENT" were removed; and tokens of type "NEWLINE", "NL", "INDENT", "DEDENT", and "ENDMARKER" were replaced with text "NEWLINE", "NL", "INDENT", "DEDENT", and "ENDMARKER" with each text attached to a space character at the start.

4.2.3 Tokenization

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. The BPE token learning algorithm is as follows [30]:

Algorithm 2 BPE token learner algorithm

function BYTE PAIR ENCODING(strings C, number of merges k)
  $V \leftarrow$ all unique characters in C ▷ initial set of tokens is characters
  for i = 1 to k **do** ▷ merge tokens til k times
    $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
    $t_{NEW} \leftarrow t_L + t_R$ ▷ make new token by concatenating
    $V \leftarrow V + t_{NEW}$ ▷ update the vocabulary
    Replace each occurrence of t_L, t_R in C with t_{NEW} ▷ and update the corpus
  end for
  return V
end function

For the transformer model, the BPE tokenization of the programs was done using fastBPE. First, the BPE codes were learned from both the Java and Python programs in the training dataset. The learned BPE codes were then applied separately to the training set of Java and Python programs. Finally, the Java and Python program BPE codes were used to construct separate Java and Python program vocabularies and additional special tokens "<unk>", "<pad>", "<bos>", and "<eos>" were added to each vocabulary. The details of the number of codes and vocabulary are shown in Table 4.2.

Table 4.2: Dataset BPE description

No. of codes	10000
Java vocabulary size	3279
Python vocabulary size	3472
Total	**6751**

For CodeBERT, RobertaTokenizer pretrained on "microsoft/codebert-base" was used. RobertaTokenizer follows the byte-level BPE approach and has a 50265-sized vocabulary.

4.3 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.

4.3.1 Transformer

An overview of the transformer model is illustrated in Figure 4.2.

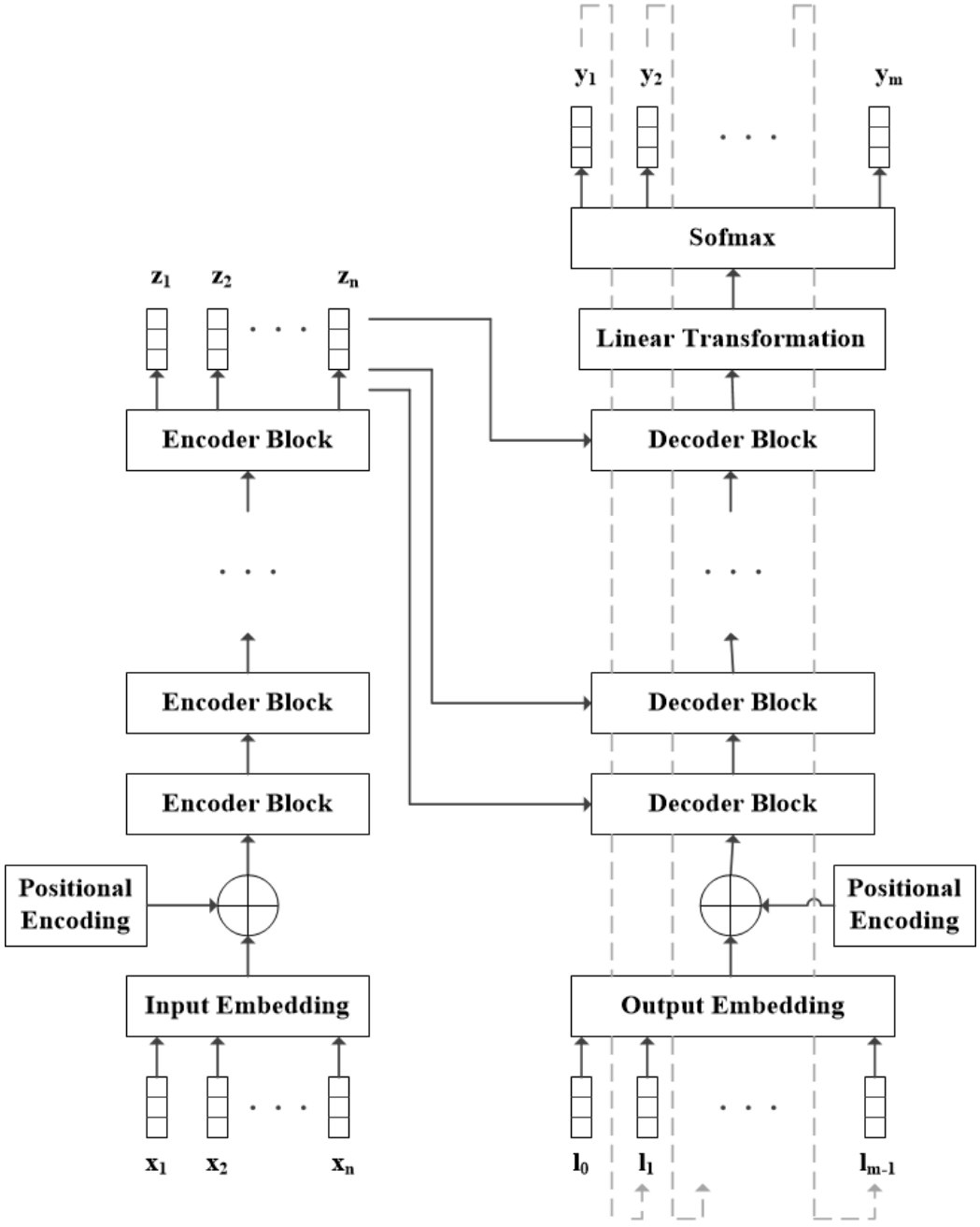


Figure 4.2: Overview of the transformer model

The following steps were used to train the model.

Step 1: Process input

The tokens of each tokenized program were converted to the vocabulary indexes.

For each batch of training data, the index of tokens "<bos>" and "<eos>" were added at the

beginning and end of the training data. And, the resulting vector was padded to the maximum length by appending "<pad>" token indexes.
The padding entry in the source data was masked to be ignored, and the future positions in the target data were masked to be hidden.

Step 2: Generate input embeddings

The inputs were first randomly initialized to the vectors of dimension d_{model} in the embedding layer, which were then updated during backpropagation.

Step 3: Generate positional embeddings

The positional encoding vector of dimension d_{model} was calculated using Equation 2.2 and Equation 2.3, which was then added to each embedding vector before applying dropout with probability p.

Step 4: Perform multi-head attention mechanism

In the multi-head attention sub-layer of the encoder, each input vector x_i was transformed to Q, K, and V. The h heads were run in parallel. After that, the outputs of h heads were concatenated and multiplied by the weight matrix W^O as given by Equation 2.5.
Inside each head, $head_i$, of the attention layer, each vector has three representations obtained by multiplying the vectors with the respective weights W_i^Q, W_i^K, and W_i^V.

i. A query vector, Q, of dimension d_k. It is trained when an input vector x_i requires all of the key-value pairs of the other input vectors, including itself.
ii. A key vector, K, of dimension d_k, it is trained to calculate attention value.
iii. A value vector, V, of dimension d_v, is trained to calculate attention value.

The attention values were computed using Equation 2.4.

Step 5: Apply post-layer normalization

The output and input of the previous sub-layer were added, and the resulting vector was normalized using $LayerNorm$ of Equation 2.8.

Step 6: Apply feed-forward network

The input and output of the feed-forward network are d_{model}, whereas the inner layer is d_{ffn}.

The output of the network was subjected to post-layer normalization.

Step 7: Repeat

Step 4 to 6 were applied N times.

Step 8: Generate output positional embeddings

In the decoder part, Step 2 and Step 3 were performed to produce output positional embeddings.

Step 9: Perform decoder multi-head attention mechanism

To keep future positions out of sight, masked multi-head attention was used on the target vector. Following that, post-layer normalization was applied to the vectors. Then, as in Step 4, the second multi-head attention mechanism was implemented, with K and V derived from the encoder output and Q derived from the prior decoder sub-layer. The results were then passed through the post-layer normalization.

Step 10: Apply decoder feed-forward network

The identical procedure as in Step 6 was followed when applying feed-forward network.

Step 11: Repeat decoder

Steps 9 and 10 were repeated N times.

Step 12: Apply linear transformation and softmax

The linear transformation was applied to the output of the decoder stack. The output was then converted into a probable element using softmax.

Step 13: Compute loss and backpropagate

The loss was computed using the cross entropy loss, and each batch was optimized using the Adam optimization algorithm.

4.3.2 CodeBERT

In the encoder-decoder model based on CodeBERT, both the encoder and the decoder were initialized with CodeBERT's pretrained weight parameters. Because CodeBERT solely consists of encoder layers, the weight parameters of the decoder layer that is not present in CodeBERT were randomly initialized. In this study, the weights of encoders were shared with those of decoders to reduce the memory usage. The overview of the CodeBERT-based encoder-decoder model is illustrated in Figure 4.3.

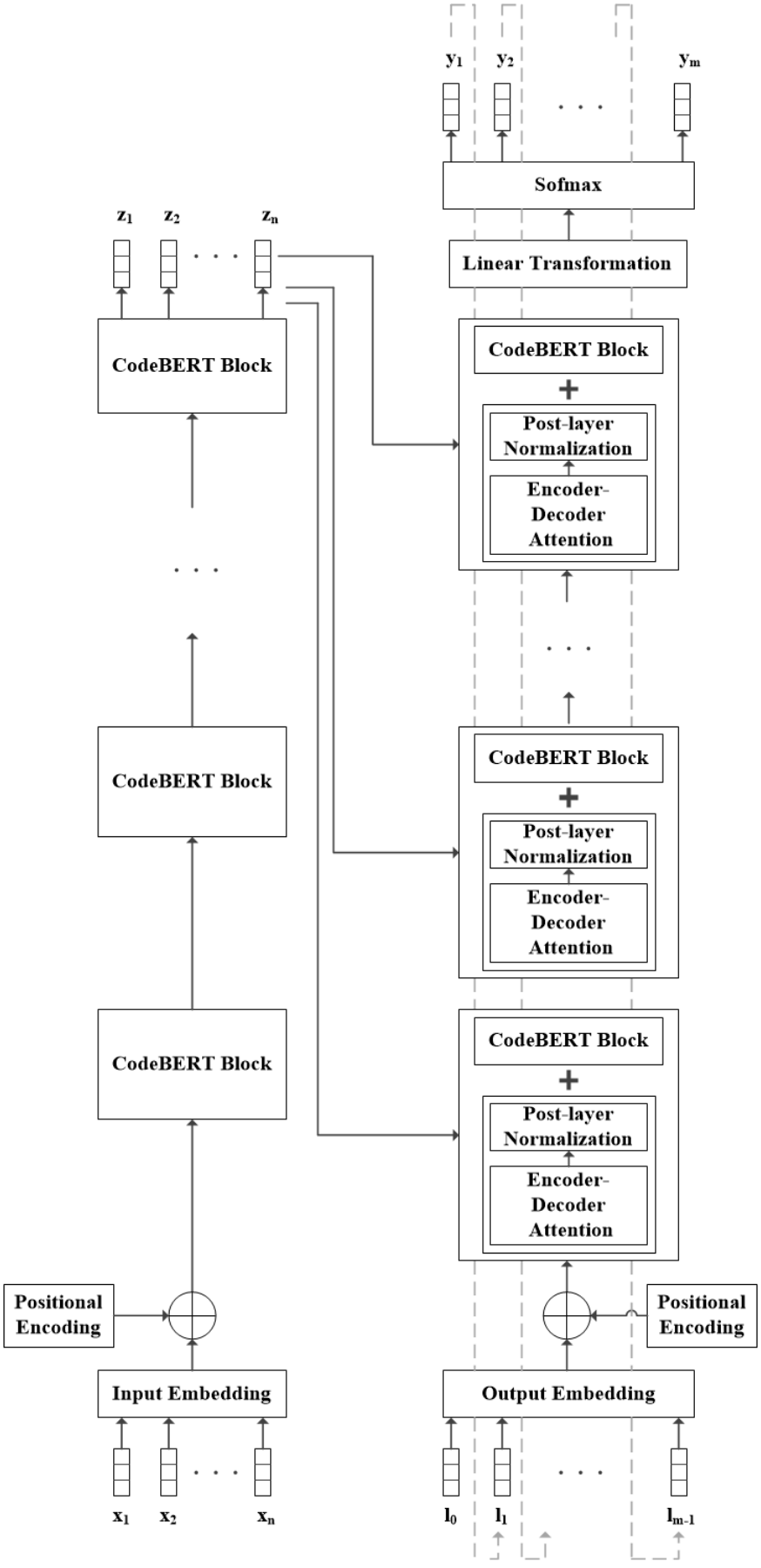


Figure 4.3: Overview of the CodeBERT model

The following steps were used to train the model.

Step 1: Initialize the encoder-decoder model

Both the encoder and the decoder of the encoder-decoder model were initialized with the pretrained "microsoft/codebert-base" model with all encoder weights tied to their equivalent decoder weights.

Step 2: Add and update configuration

The configuration of the model, such as "decoder_start_token_id", "eos_token_id", "pad_token_id", "max_length", "no_repeat_ngram_size", and "vocab_size", were added and updated.

Step 3: Train model

The configured model was trained following the same steps (2-13) as the transformer model, with the exception that the weights were shared and were taken from the pretrained model. During backpropagation, the shared weights were updated twice: once in the decoder stack and then in the encoder stack.

4.4 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. In other words,

$$x_t = argmax_{\tilde{x}_t} \mathbb{P}(\tilde{x}_t | x_1, \ldots, x_n) \tag{4.1}$$

This technique is efficient and natural. However, it explores only a small part of the search space [19].

4.5 Data Postprocessing

The programs translated using the transformer were postprocessed by first removing BPE tokens and "<unk>" tokens. In the case of the 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.

For model evaluation, both the predicted and reference Python programs were detokenized by splitting the program on a single space character. The “NL”, “DEDENT”, “ENDMARKER”, and multiple “NEWLINE” tokens were removed and the “NEWLINE” and “INDENT” were replaced with “\\n” and “\\t” respectively. The tokens were then joined with a single space character.

For programs translated using CodeBERT, the same data postprocessing steps were followed, but BPE special token and the “<unk>” token were not replaced.

4.6 Evaluation

The BLEU score and the CodeBLEU score were used to assess the transformer and CodeBERT models’ performance. Both the BLEU and CodeBLEU scores range from 0 to 1, with 0 indicating a perfect mismatch and 1 indicating a perfect match.

CHAPTER 5

IMPLEMENTATION

Throughout this study, Python was used as the programming language. The transformer-based neural translation model was implemented using PyTorch, and the CodeBERT-based encoder-decoder model was initialized using HuggingFace.

5.1 Implementation Tools

The following are the tools employed in the study:

i. Python programming language

Python is a high-level, interpreted, and dynamically-typed programming language, first released in 1991 by Guido van Rossum. It supports procedural, functional, and object-oriented programming. Python is the most often used language for creating machine learning projects. In the study, the Anaconda distribution of Python version 3.9 was used on a personal computer, and the default version 3.7 was used in Colab and Kaggle.

ii. Anaconda distribution

Anaconda distribution is an open-source Python distribution platform for scientific computing such as data science and machine learning. It allows users to effortlessly install, run, and update packages using Conda, an open-source package management system. An Anaconda distribution version of 2022.05 was used.

iii. Jupyter notebook

Jupyter notebook is an open-source web-based tool used for creating computational notebooks. The kernel embedded in the notebook handles the execution of code. The Jupyter notebook of version 6.4.8 provided by Anaconda distribution was used for this study.

iv. Google colaboratory (Colab)

Colab is a free Jupyter notebook environment provided by Google Research. It allows Google users to write and execute Python code without any setup. Colab has libraries like PyTorch and TensorFlow integrated within its environment. It provides 12GB of RAM, an Nvidia K80/T4 GPU with 12GB/16GB of GPU memory, and 358GB of disk space

when the GPU is enabled. It has a 12 hour maximum execution time and a 90 minute maximum idle time [31]. Colab was used to train the models, to perform translation, and to evaluate the models.

v. Kaggle kernel

Kaggle kernel is a free platform that allows its users to run Jupyter notebook in their browser without any configuration and installation of framework like PyTorch. It provides an Nvidia P100 GPU with 16GB of GPU memory, 12GB of RAM, and 5GB of disk space. The maximum execution time is 9 hours and the maximum idle time is 60 minutes [31]. However, Kaggle kernel requires the user's phone number to connect the virtual machine to the internet. It was used to train the neural translation models in this study.

vi. Javalang, Tokenize, and FastBPE

Javalang is a Python library built on the Java 8 language specification. It provides a lexer and a parser for Java programs.

The module, tokenize, is a lexical scanner implemented in Python. When given the Python source code, this module returns the tokens of the Python code.

FastBPE is the implementation of the Byte Pair Encoding algorithm in C++ with a Python API.

vii. PyTorch

PyTorch is a framework built on top of the open-source machine learning library, Torch. It was developed by Meta AI to be used for applications like natural language processing and computer vision. PyTorch has a Python and C++ interface with two high-level features: Tensor computing using a Graphical Processing Unit (GPU) and Deep Neural Networks built on an automatic differentiation system. For the study, PyTorch version 1.11.0 was used.

viii. HuggingFace

HuggingFace is a data science platform that provides tools to build, train, and deploy machine learning models. It provides a platform to store and share machine learning models and datasets. The HuggingFace repository contains trained models such as BERT, RoBERTa, CodeBERT, and so on, which can be used by anyone.

ix. CodeXGLUE

CodeXGLUE, General Language Understanding Evaluation benchmark for CODE, is a set of code intelligence tasks as well as a platform for evaluating and comparing the models. It is equipped with the CodeBLEU evaluation mechanism. The CodeBLEU implementation in CodeXGLUE was used to assess program translation models.

x. Matplotlib

Matplotlib is a cross-platform data visualization library. The object-oriented APIs provided by matlotlib can be used to integrate plots in applications. The matplotlib library can be used in Jupyter notebook and other web application servers.

5.2 Test Environment

The hardware and software specifications used in the study are listed below.

Hardware Specifications

- Processor: 11th Gen Intel(R) Core(TM) i7-11390H @ 3.40GHz
- RAM: 16GB
- Disk Space: 500GB SSD

Software Specifications

- Operating System: Microsoft Windows 10 Home
- Programming Language: Python
- Tools: Jupyter Notebook, Google Colaboratory, Kaggle Kernel

5.3 Hyperparameters

The following are the hyperparameters that were used in the study:

i. No. of layers

It refers to the number of encoder and decoder layers used in the model. 6 encoder layers and 6 decoder layers are the default settings.

ii. No. of heads

In a multi-headed attention mechanism, this is the number of attention heads used. The number of heads is set to 8 by default.

iii. Embedding size

The embedding size is the dimension of each token's vector, and it is set to 512 by default.

iv. Feed-Forward network hidden dimension

It is the dimension of the inner layer of the feed forward network model. The value is 2048 by default.

v. Activation

An activation function is a function that transforms the weighted sum of inputs into an output. The default activation function is ReLU.

vi. Dropout

Dropout is a technique where randomly selected values are zeroed out. The default dropout is 0.1.

vii. Layer normalization epsilon

To avoid division by zero, a small float value is added to variance. 1e-5 is the default value.

viii. Loss function

A loss function is a method for assessing how effectively a model works. The study used a cross entropy loss function.

ix. Optimizer

An optimizer is an algorithm that adjusts the weights and the learning rate to reduce the overall loss. By default, the Adam optimizer is used.

x. Batch size

It is the number of training instances used in a single iteration.

xi. Learning rate

The learning rate is defined as a tuning parameter that controls a step size in each iteration to minimize the loss.

xii. Number of epochs

It refers to the number of times the entire training dataset is fed into the model.

xiii. α, β, γ, δ (CodeBLEU)

These are the weights given to each component in the calculation of the CodeBLEU score. The default values are 0.25, 0.25, 0.25, and 0.25.

The values of the hyperparameters used to train the models are shown in the Table 5.1.

Table 5.1: Hyperparameters

Hyperparameter	**Transformer**	**CodeBERT**
No. of layers	(6, 6), (12, 12)	(6, 6), (12, 12)
No. of heads	12	12
Embedding size	768	768
Feed-Forward network hidden dimension	3072	3072
Activation	GELU	GELU
Dropout	0.1	0.1
Layer normalization epsilon	1e-12	1e-12
Loss function	Cross Entropy Loss	Cross Entropy Loss
Optimizer	Adam Optimizer	Adam Optimizer
Batch size	16	16
Learning rate	2e-5	2e-5
No. of epochs	50, 100	50, 100

CHAPTER 6

RESULTS AND ANALYSIS

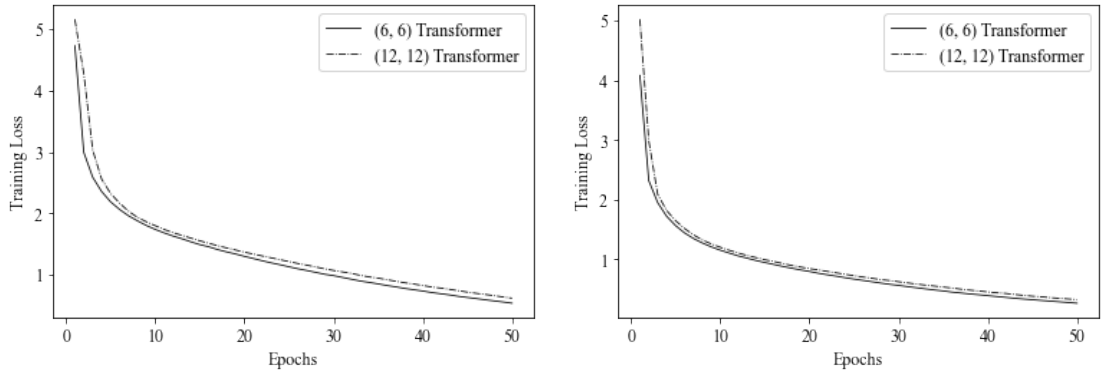
6.1 Training Loss

The training loss is a measure that is used to determine how well a deep learning model fits the training data. The cross entropy loss was used to compute the loss of the transformer and the CodeBERT model. The experiment was conducted on different configurations of the transformer and the CodeBERT model, for varying numbers of epochs.

Case I: 50 epochs

Transformer

The following graphs illustrate the pattern of training loss for transformers with (6, 6) layers of encoder and decoder and (12, 12) layers of encoder and decoder.



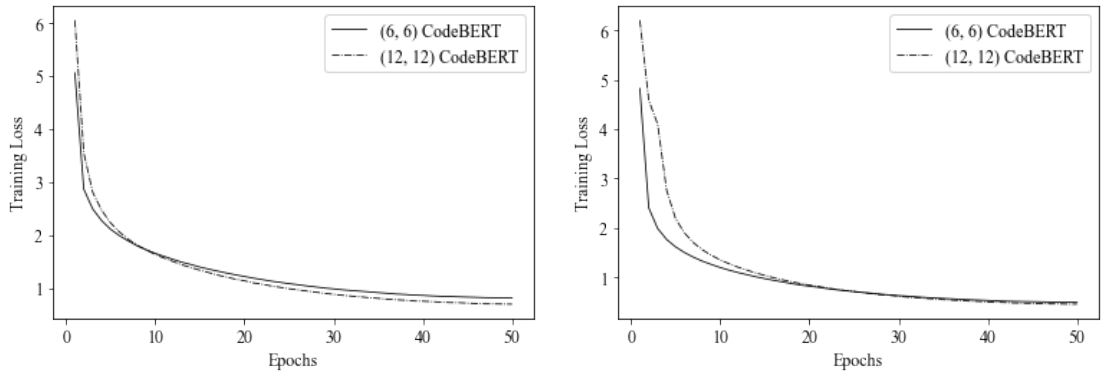
(a) Java to Python Translation *(b) Python to Java Translation*

Figure 6.1: Training loss of the transformer model with 6 encoder layers and 6 decoder layers, and 12 encoder layers and 12 decoder layers, for 50 epochs

The training losses of 0.5366 and 0.6188 were obtained for the transformer model with (6, 6) and (12, 12) layers, respectively, while training the models to translate from Java to Python, as shown in Figure 6.1a. In the case of Python to Java translation (Figure 6.1b), losses of 0.2692 and 0.3295 were observed for (6, 6) and (12, 12) layers, respectively.

CodeBERT

The training losses of the models initialized with (6, 6) and (12, 12) layers of CodeBERT are shown in Figure 6.2.



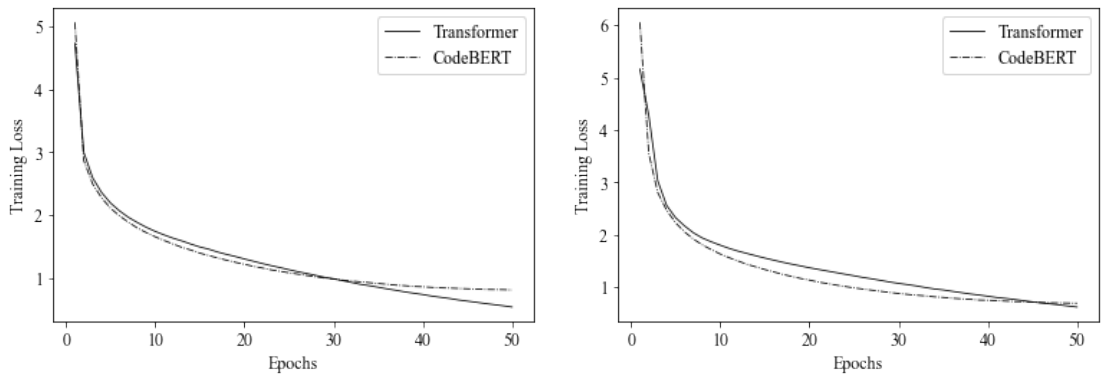
(a) Java to Python Translation *(b) Python to Java Translation*

Figure 6.2: Training loss of the CodeBERT model with 6 encoder layers and 6 decoder layers, and 12 encoder layers and 12 decoder layers, for 50 epochs

For Java to Python translation, the losses of 0.8076 and 0.6914 were obtained for (6, 6) and (12, 12) layers, respectively; and for Python to Java translation, the losses of 0.4945 and 0.4588 were obtained, respectively.

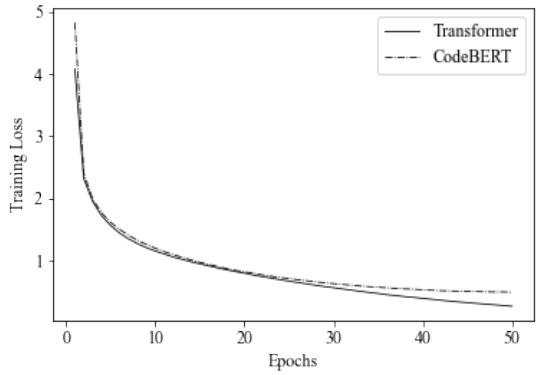
Transformer vs CodeBERT

Figure 6.3 shows the transformer and CodeBERT training losses of 0.5366 and 0.8076 for (6, 6) layers and 0.6188 and 0.6914 for (12, 12) layers when trained to translate from Java to Python; and Figure 6.4 shows the losses of 0.2692 and 0.4945 for (6, 6) layers and 0.3295 and 0.4588 for (12, 12) layers of the models when trained to translate from Python to Java.

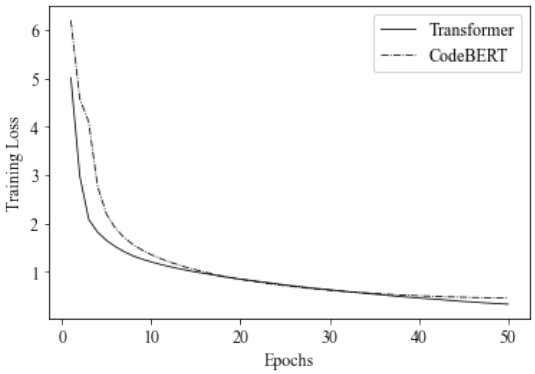


(a) Training loss of the models with 6 encoder layers and 6 decoder layers *(b) Training loss of the models with 12 encoder layers and 12 decoder layers*

Figure 6.3: Training loss of the transformer and CodeBERT model in Java to Python translation for 50 epochs



(a) Training loss of the models with 6 encoder layers and 6 decoder layers



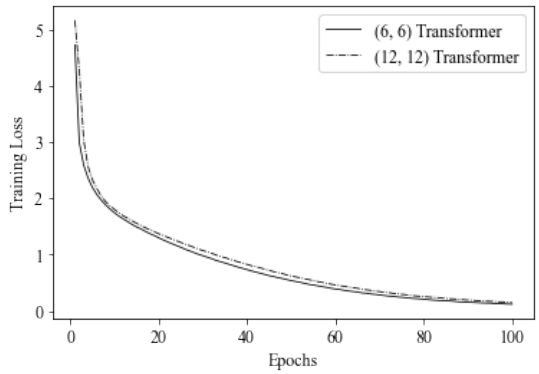
(b) Training loss of the models with 12 encoder layers and 12 decoder layers

Figure 6.4: Training loss of the transformer and CodeBERT model in Python to Java translation for 50 epochs

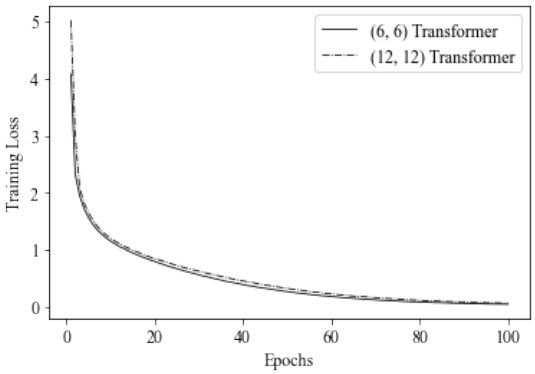
Case II: 100 epochs

Transformer

Figure 6.5a depicts the training loss for (6, 6) and (12, 12) transformers while training from Java to Python; the final losses are 0.1176 and 0.1461, respectively. Similarly, Figure 6.5b shows the losses of 0.0515 and 0.0697 observed in models while training in the Python to Java direction.



(a) Java to Python Translation

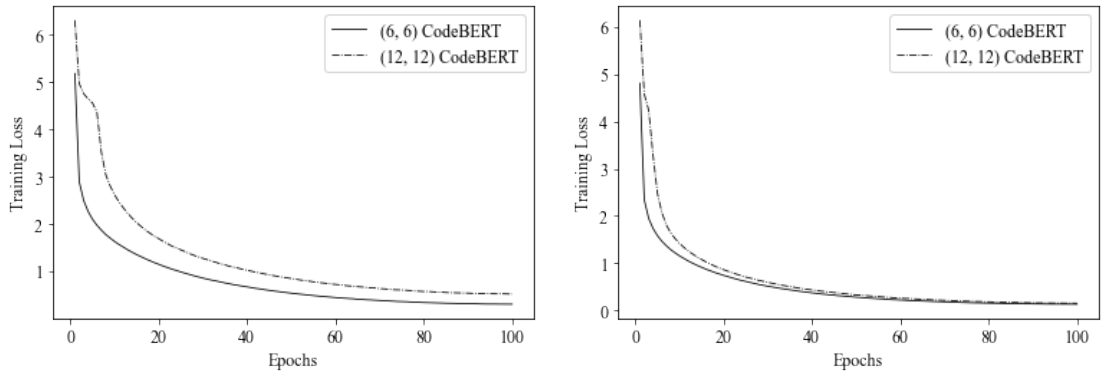


(b) Python to Java Translation

Figure 6.5: Training loss of the transformer model with 6 encoder layers and 6 decoder layers, and 12 encoder layers and 12 decoder layers, for 100 epochs

CodeBERT

Figure 6.6 shows the loss for (6, 6) and (12, 12) CodeBERT models trained in Java to Python direction; the losses are 0.3088 and 0.5233, respectively. In Java to Python direction, the losses observed are 0.1381 and 0.1575, respectively.

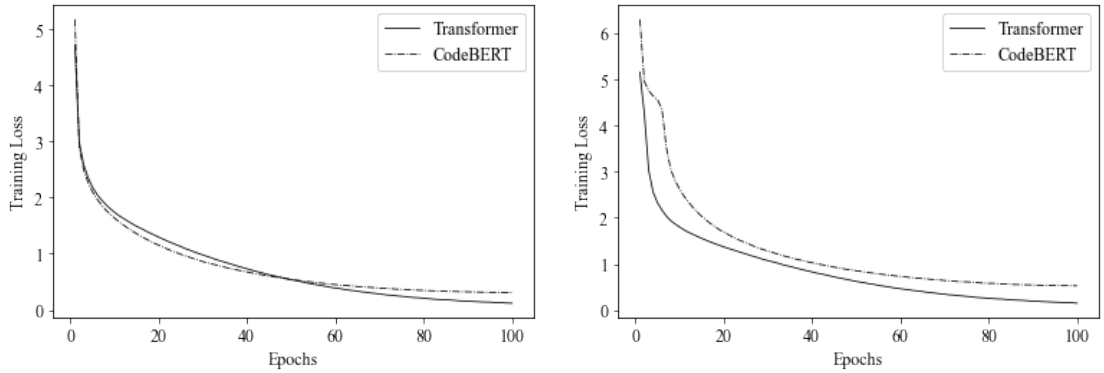


(a) Java to Python Translation *(b) Python to Java Translation*

Figure 6.6: Training loss of the CodeBERT model with 6 encoder layers and 6 decoder layers, and 12 encoder layers and 12 decoder layers, for 50 epochs

Transformer vs CodeBERT

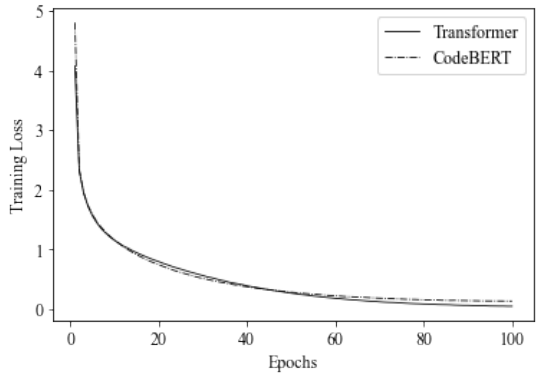
The training losses of (6, 6) and (12, 12) layers transformer and CodeBERT models in Java to Python translation direction are shown in Figure 6.7. For (6, 6) layers the loss at the end of epoch for the transformer model is 0.1176 and for the CodeBERT model is 0.3088. For (12, 12) layers the loss for the tranformer model is 0.1461 and the CodeBERT model is 0.5233.



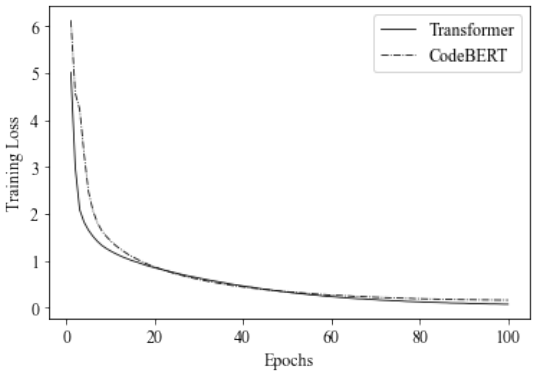
(a) Training loss of the models with 6 encoder layers and 6 decoder layers *(b) Training loss of the models with 12 encoder layers and 12 decoder layers*

Figure 6.7: Training loss of the transformer and CodeBERT model in Java to Python translation for 100 epochs

Similarly, the training losses for the transformer and CodeBERT models in Python to Java translation direction are show in Figure 6.8. The transformer and CodeBERT model with (6, 6) layers have training loss 0.0514, 0.1381 respectively, and with (12, 12) layers have 0.0697, 0.1575, respectively.



(a) Training loss of the models with 6 encoder layers and 6 decoder layers



(b) Training loss of the models with 12 encoder layers and 12 decoder layers

Figure 6.8: Training loss of the transformer and CodeBERT model in Python to Java translation for 100 epochs

6.2 Results

The BLEU and CodeBLEU scores of the transformer and CodeBERT models obtained for 627 testing samples are shown in Table 6.1 (Java to Python) and Table 6.2 (Python to Java).

Table 6.1: BLEU, CodeBLEU, WNM, SM, and DM scores for Java to Python translation

No. of epochs	No. of layers	Model	BLEU	CodeBLEU	WNM	SM	DM
50	6	Transformer	**0.2812**	**0.2802**	**0.2828**	0.3137	0.2430
		CodeBERT	0.1294	0.2109	0.1321	**0.3491**	0.2334
	12	Transformer	0.2627	0.2724	0.2642	0.3227	0.2400
		CodeBERT	0.0977	0.1897	0.1028	0.3439	0.2152
100	6	Transformer	0.2659	0.2733	0.2683	0.3097	**0.2492**
		CodeBERT	0.1512	0.2233	0.1539	0.3485	0.2401
	12	Transformer	0.2519	0.2656	0.2536	0.3154	0.2415
		CodeBERT	0.1067	0.1745	0.1092	0.3063	0.1760

Table 6.2: BLEU, CodeBLEU, WNM, SM, and DM scores for Python to Java translation

No. of epochs	No. of layers	Model	BLEU	CodeBLEU	WNM	SM	DM
50	6	Transformer	0.3883	0.4016	0.3990	0.4715	0.3434
		CodeBERT	0.2747	0.3495	0.3006	0.4857	0.3234
	12	Transformer	0.3633	0.3841	0.3744	0.4591	0.3358
		CodeBERT	0.2580	0.3299	0.2770	0.4644	0.3112
100	6	Transformer	**0.3891**	**0.4018**	**0.4012**	0.4613	**0.3526**
		CodeBERT	0.2938	0.3727	0.3203	**0.5172**	0.3458
	12	Transformer	0.3652	0.3837	0.3778	0.4461	0.3434
		CodeBERT	0.2936	0.3697	0.3183	0.5120	0.3424

6.3 Analysis

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 transformer model with 6 layers has a lower training loss than the model with 12 layers over a period of 50 epochs. Both the 6-layer Java to Python transformer translation model and the 6-layer Python to Java transformer translation model have losses of 0.5366 and 0.2692, respectively. In the case of the CodeBERT models trained for 50 epochs, the 12-layer models exhibit less loss than the 6-layer models. The loss for the 12-layer Java to Python CodeBERT model is 0.6914, and the loss for the Python to Java CodeBERT model is 0.4588. In comparison to CodeBERT models with 6 layers, transformer models with 6 layers produce lower training loss. The 6-layer transformer model has a loss of 0.5366 and 0.2692 when training from Java to Python and Python to Java, respectively. The 12-layer transformer also performs better than the 12-layer CodeBERT in terms of training loss, with losses of 0.3295

and 0.6188 when training from Python to Java and Java to Python, respectively. Even for 100 epochs, the 6-layer transformer models have less training loss than the 12-layer transformer models. Likewise, 6-layer CodeBERT models show less loss than 12-layer CodeBERT models after 100 training epochs. For 100 training epochs, the CodeBERT models also suffer from high training loss in comparison to the transformer models.

The results shown in Table 6.1 demonstrate 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 0.2812 and 0.2802, respectively. According to the results shown in Table 6.2, the transformer model, trained for 100 epochs to translate from Python to Java, achieved the highest BLEU and CodeBLEU scores, respectively, of 0.3891 and 0.4018. This model also has 6 encoder and 6 decoder layers. 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. In addition to this, the transformer models have achieved higher WNM, and DM scores than the CodeBERT models. However, the SM scores of the transformer models are lower than those of the CodeBERT models.

CHAPTER 7

CONCLUSION

7.1 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. To be able to translate the programs, the study followed the following steps: data preprocessing, model training, inference, and data postprocessing.

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 0.2812 and 0.2802, 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 0.3891 and 0.4018, respectively. Furthermore, the scores of Java to Python translation models differ from those of Python to Java translation models.

7.2 Limitations

The study on program translation has the following limitations:

- The autoregressive model pretrained on codes exists. However, the study focused on exploiting the public CodeBERT checkpoint to initialize the encoder and the decoder of the encoder-decoder model.
- The parallel corpora of other programming languages, such as Java-C++, Java-C#, PHP-Java, etc., are also available, but the study used only the Java-Python corpus to train and evaluate the models.
- Despite the availability of evaluation metrics like computational accuracy, the dissertation has focused on using BLEU and CodeBLEU scores, which are the most extensively used

evaluation metrics for assessing the performance of the translation models.

7.3 Future Recommendations

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

[1] I. Sommerville, "Source code translation," in *Software Engineering*, 6th ed. Addison Wesley, 2002, ch. 28, pp. 573–574. [Online]. Available: https://ifs.host.cs.st-andrews.ac.uk/Resources/Notes/Evolution/SWReeng.pdf.

[2] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

[3] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," *arXiv preprint arXiv:2108.11590*, 2021.

[4] M.-Y. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," unpublished.

[5] M. Taverna, "Language2language transformers: Machine learning to build transpilers." tomassetti.me. https : / / tomassetti . me / language2language - transformers - machine - learning-to-build-transpilers/ (accessed May 14, 2022).

[6] B. L. Solutions, "Everything you need to know about neural machine translation." biglanguage.com. https://biglanguage.com/blog/everything-to-know-about-neural-machine-translation (accessed May 14, 2022).

[7] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] J. Villa and Y. Zimmerman, "Warm starting for efficient deep learning resource utilization." determined.ai. https://www.determined.ai/blog/warm-starting-deep-learning (accessed May 14, 2022).

[9] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting python 2 to python 3 code," *PeerJ PrePrints*, 2015.

[10] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in neural information processing systems*, vol. 31, 2018.

[11] O. Technologies, “When, why and how to migrate from statistical to neural machine.” omniscien.com. https://omniscien.com/blog/migrate-from-smt-to-nmt (accessed May 22, 2022).

[12] D. Rothman, *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*. Packt Publishing Ltd, 2021.

[13] J. Alammar, “The illustrated transformer.” jalammar.github.io. https://jalammar.github.io/illustrated-transformer/ (accessed May 22, 2022).

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[15] C. Durgia, “Exploring bert variants (part 1): Albert, roberta, electra.” towardsdatascience.com. https://towardsdatascience.com/exploring-bert-variants-albert-roberta-electra-642dfe51bc23 (accessed May 22, 2022).

[16] Z. Feng *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.

[17] A. I. Magazine, “Microsoft Introduces First Bimodal Pre-Trained Model for Natural Language Generation.” analyticsindiamag.com. https://analyticsindiamag.com/microsoft-introduces-first-bimodal-pre-trained-model-for-natural-language-generation (accessed May 22, 2022).

[18] C. Hansen, “Activation functions explained - gelu, selu, elu, relu and more.” mlfromscratch.com. https://mlfromscratch.com/activation-functions-explained/ (accessed Jun. 25, 2022).

[19] C. Manning, R. Socher, G. G. Fang, and R. Mundra, “Cs224n: Natural language processing with deep learning.” web.stanford.edu. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194.

[20] A. Ng, “Optimization algorithms.” DeepLearning.AI. https://cs230.stanford.edu/files/C2M2.pdf (accessed Jun. 25, 2022).

[21] S. Ren *et al.*, “Codebleu: A method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.

[22] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Lexical statistical machine translation for language migration,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 651–654.

[23] M. H. Hassan, O. A. Mahmoud, O. I. Mohammed, A. Y. Baraka, A. T. Mahmoud, and A. H. Yousef, “Neural machine based mobile applications code translation,” in *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, IEEE, 2020, pp. 302–307.

[24] S. Rothe, S. Narayan, and A. Severyn, “Leveraging pre-trained checkpoints for sequence generation tasks,” *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 264–280, 2020.

[25] M. Tauda, Z. Zainuddin, and Z. Tahir, “Programming language translator for integration client application with web apis,” in *2021 International Conference on Artificial Intelligence and Mechatronics Systems (AIMS)*, IEEE, 2021, pp. 1–4.

[26] D. Feng and S. Chiba, “Attempts on using syntax trees to improve programming language translation quality by machine learning,” Graduate School of Information Science and Technology, The University of Tokyo, Technical Report, 2021. [Online]. Available: https://static.csg.ci.i.u-tokyo.ac.jp/papers/21/daifeng-jssst2021.pdf.

[27] M. Shah, R. Shenoy, and R. Shankarmani, “Natural language to python source code using transformers,” in *2021 International Conference on Intelligent Technologies (CONIT)*, IEEE, 2021, pp. 1–4.

[28] E. Mashhadi and H. Hemmati, “Applying codebert for automated program repair of java simple bugs,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 505–509.

[29] A. Karmakar and R. Robbes, “What do pre-trained code models know about code?” In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1332–1336.

[30] D. Jurafsky and J. H. Martin, “Basic text processing.” web.stanford.edu. https://web.stanford.edu/~jurafsky/slp3/slides/2_TextProc_Jan_06_2021.pdf (accessed Jun. 20, 2022).

[31] A. Kazemnejad, “How to do deep learning research with absolutely no gpus - part 2.” kazemnejad.com. https://kazemnejad.com/blog/how_to_do_deep_learning_research_with_absolutely_no_gpus_part_2/ (accessed Jun. 20, 2022).

APPENDIX A

SAMPLE DATASET

Sample 1:

Java Program

```
// Java implementation to find
// the sum of the given series
import java.io.*;

class GfG {

// function to find the sum
// of the given series
static int sumOfTheSeries(int n)
{
    // required sum
    return (n * (n + 1) / 2) *
            (2 * n + 1) / 3;
}



// Driver program to test above
public static void main (String[] args)
{
    int n = 5;

    System.out.println("Sum = "+
                sumOfTheSeries(n));

}

}
```

```
// This code is contributed by Gitanjali.
```

Python Program

```
# Python3 implementation to find
# the sum of the given series

# functionn to find the sum
# of the given series
def sumOfTheSeries( n ):

    # required sum
    return int((n * (n + 1) / 2) *
            (2 * n + 1) / 3)

# Driver program to test above
n = 5
print("Sum =", sumOfTheSeries(n))

# This code is contributed by "Sharad_Bhardwaj".
```

Sample 2:

Java Program

```
/*
 * Solution to Project Euler problem 7
 * Copyright (c) Project Nayuki. All rights reserved.
 *
 * https://www.nayuki.io/page/project-euler-solutions
 * https://github.com/nayuki/Project-Euler-solutions
 */


public final class p007 implements EulerSolution {

	public static void main(String[] args) {
		System.out.println(new p007().run());
	}


	/*
	 * Computers are fast, so we can implement this solution by
	 ↪ testing each number
	 * individually for primeness, instead of using the more
	 ↪ efficient sieve of Eratosthenes.
	 */
	public String run() {
		for (int i = 2, count = 0; ; i++) {
			if (Library.isPrime(i)) {
				count++;
				if (count == 10001)
					return Integer.toString(i);
			}
		}
```

```
        }

}
```

Python Program

```
#
# Solution to Project Euler problem 7
# Copyright (c) Project Nayuki. All rights reserved.
#
# https://www.nayuki.io/page/project-euler-solutions
# https://github.com/nayuki/Project-Euler-solutions
#

import eulerlib, itertools



# Computers are fast, so we can implement this solution by testing each
 ↪  number
# individually for primeness, instead of using the more efficient sieve
 ↪  of Eratosthenes.
#
# The algorithm starts with an infinite stream of incrementing integers
 ↪  starting at 2,
# filters them to keep only the prime numbers, drops the first 10000
 ↪  items,
# and finally returns the first item thereafter.
def compute():
        ans = next(itertools.islice(filter(eulerlib.is_prime,
         ↪  itertools.count(2)), 10000, None))
        return str(ans)
```

```
if __name__ == "__main__":
        print(compute())
```

APPENDIX B

EXAMPLE TRANSLATION

A single Java-Python parallel program from the test dataset is:

Java Program

```
// Java program to find sum of all elements
// upto nth row in Pascal triangle.
import java.io.*;

class GFG {

    // Function to find sum of aal elements
    // upto nth row.
    static long calculateSum(int n)
    {

        // Initialize sum with 0
        long sum = 0;

        // Loop to calculate power of 2
        // upto n and add them
        for (int row = 0; row < n; row++) {
            sum = sum + (1 << row);
        }

        return sum;
    }

    // Driver code
    public static void main(String[] args)
    {
        int n = 10;
```

```
        System.out.println("Sum of all elements:"
                           + calculateSum(n));
    }
}
```

Python Program

```
# Python program to find sum of all elements
# upto nth row in Pascal triangle.

# Function to find sum of aal elements
# upto nth row.
def calculateSum(n) :

    # Initialize sum with 0
    sum = 0

    # Loop to calculate power of 2
    # upto n and add them
    for row in range(n):
        sum = sum + (1 << row)

    return sum

# Driver code
n = 10
print("Sum of all elements:", calculateSum(n))
```

The following is an illustration of the outputs generated by the steps in the translation process:

Case I: Java to Python translation using the transformer model with 6 encoder and 6 decoder layers trained for 50 epochs

i. Pretokenization

```
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 ) ) ; } }
```

ii. Tokenization

```
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 ) ) ; }
```

iii. Translation

```
def calculateSum ( n ) : NEWLINE INDENT sum = 0 ; NEWLINE for i in
↪ range ( n ) : NEWLINE INDENTINDENT sum = sum + ( 1 << i ) ;
↪ NEWLINE DEDENT return sum ; NEWLINE DEDENT n = 10 ; NEWLINE
↪ print ( "Sum of all is" , calculateSum ( n ) ) NEWLINE NL
↪ ENDMARKER
```

iv. Data Postprocessing

```
def calculateSum(n):
 sum=0;
for i in range(n):
 sum=sum+(1<<i);
return sum;
```

```
n=10;
print("Sum of all is",calculateSum(n))
# Created by pyminifier (https://github.com/liftoff/pyminifier)
```

Case II: Java to Python translation using the CodeBERT model with 6 encoder and 6 decoder layers trained for 100 epochs

i. Pretokenization

```
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)); } }
```

ii. Tokenization

```
import Ġjava . io . Ġ* Ġ; Ġclass ĠG FG Ġ{ Ġstatic Ġlong Ġcalculate
↪ Sum ( int Ġn ) Ġ{ Ġlong Ġsum Ġ= Ġ0 ; Ġfor ( int Ġrow Ġ= Ġ0 ;
↪ Ġrow Ġ< Ġn ; Ġrow Ġ++ Ġ) Ġ{ Ġsum Ġ= Ġsum Ġ+ Ġ( 1 Ġ<< Ġrow ); Ġ}
↪ Ġreturn Ġsum ; Ġ} Ġpublic Ġstatic Ġvoid Ġmain ( String [] args
↪ ) Ġ{ Ġint Ġn Ġ= Ġ10 ; ĠSystem . out . println (" Sum Ġof Ġall
↪ Ġelements :" Ġ+ Ġcalculate Sum ( n )); Ġ} Ġ}
```

iii. Translation

```
def calculateSum(n): NEWLINE INDENT sum=0 NEWLINE for i in
↪ range(0,n+1): NEWline INDENTINDENT sum+=i NEWLINE DEDENT
↪ return sum NEWLINE d=sum-1 NEWLINE print("Sum of
↪ triangular",calculateSum(num)) NEWLINE NL ENDMARKER
```

iv. Data Postprocessing

```
def calculateSum(n):
    sum=0
for i in range(0,n+1):
```

```
        sum+=i
 return sum
d=sum-1
print("Sum of triangular",calculateSum(num))
```

Case III: Python to Java translation using the transformer model with 6 encoder and 6 decoder layers trained for 100 epochs

i. Data Cleaning

```
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))
# Created by pyminifier (https://github.com/liftoff/pyminifier)
```

ii. Pretokenization

```
def calculateSum ( n ) : NEWLINE INDENT sum = 0 NEWLINE for row in
↪  range ( n ) : NEWLINE INDENTINDENT sum = sum + ( 1 << row )
↪  NEWLINE DEDENT return sum NEWLINE DEDENT n = 10 NEWLINE print (
↪  "Sum of all elements:" , calculateSum ( n ) ) NEWLINE NL
↪  ENDMARKER
```

iii. Tokenization

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

iv. Translation

```
import java . io . * ; class GFG { static long calculateSum ( int n
↪ ) { long sum = 0 ; for ( int i = 0 ; i < n ; i ++ ) { sum = sum
↪ + ( 1 << i ) ; } return sum ; } public static void main (
↪ String [ ] args
```

v. Data Postprocessing

```
import java.io. * ;
class GFG {
    static long calculateSum(int n) {
        long sum = 0;
        for(int i = 0;
        i < n;
        i ++ ) {
            sum = sum + (1 << i);

        }
        return sum;

    }
    public static void main(String[]args
```

Case IV: Python to Java translation using the CodeBERT model with 6 encoder and 6 decoder layers trained for 100 epochs

i. Data Cleaning

```
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))
# Created by pyminifier (https://github.com/liftoff/pyminifier)
```

ii. Pretokenization

```
def calculateSum(n): NEWLINE INDENT sum=0 NEWLINE for row in
↪ range(n): NEWLINE INDENTINDENT sum=sum+(1<<row) NEWLINE DEDENT
↪ return sum NEWLINE DEDENT n=10 NEWLINE print("Sum of all
↪ elements:",calculateSum(n)) NEWLINE NL ENDMARKER
```

iii. Tokenization

```
def Ġcalculate Sum ( n ): ĠNEW LINE ĠIND ENT Ġsum = 0 ĠNEW LINE
↪ Ġfor Ġrow Ġin Ġrange ( n ): ĠNEW LINE ĠIND ENT IND ENT Ġsum =
↪ sum +( 1 << row ) ĠNEW LINE ĠD ED ENT Ġ Ġreturn Ġsum ĠNEW LINE
↪ ĠD ED ENT Ġn = 10 ĠNEW LINE Ġprint (" Sum Ġof Ġall Ġelements :
↪ ", cal cul ate Sum ( n )) ĠNEW LINE ĠNL ĠEND M ARK ER
```

iv. Translation

```
import java.io. * ; class GFG { static long calculateSum(int n) {
↪ long sum = 0; for(int i = 0, sum = 1; i < n; i ++ )sum += (1 <<
↪ i); return sum; } public static void main(String[]args) { int n
↪ = 10; System.out.println("Sum of the series : " + calculateSum
↪ of n + " = " + sum(n)); } }
```

v. Data Postprocessing

```
import java.io. * ;
class GFG {
    static long calculateSum(int n) {
        long sum = 0;
        for(int i = 0, sum = 1;
        i < n;
        i ++ )sum += (1 << i);
```

```
        return sum;


    }
    public static void main(String[]args) {
        int n = 10;
        System.out.println("Sum of the series : " + calculateSum of
        ↪  n + " = " + sum(n));


    }
}
```