11711 - Advanced NLP Assignment 3
Code Translation: A Transformer-based Alignment Approach for Enabling Code Translation using Frozen Models

Arjun Choudhry, Piyush Khanna, Siddharth Gandhi
Master of Science in Intelligent Information Systems
Language Technologies Institute, Carnegie Mellon University
{arjuncho, piyushkh, ssg2}@andrew.cmu.edu

Abstract

In this work, we report the details of our work done towards Assignment 3 for the course 11711 - Advanced NLP. Our work is on code translation and we have done a comprehensive literature survey for the same, replicated the results of a baseline model with reasonable computational requirements, and have done thorough error analysis on the model, finding various drawbacks of the current state of code translation models. Towards this, we propose our own approach which we are working on for Assignment 4, and also include preliminary details about a dataset we are curating using code solutions available on CodeForces, which will potentially be used to improve our proposed model. We make our code and results obtained on the baseline model available here[1].

1 Introduction

Code translation is the task of automatically translating source code from a given programming language, say C++, to another desired programming language, say Python, with the constraint that the code be executable and output the same results as in the original coding language. This task is also commonly known as transcompiler or transpiler. There are a variety of use cases where this task is useful, ranging from omnipresent languages like Java, C++, and Python to new and task-specific languages like Swift and Dart and vice versa, to migrating code from legacy languages like COBOL to modern-day ones like Java and Python.

In recent years, various methods for machine translation have been applied to the task of source code translation, with initial works treating code translation as yet another common machine translation task. Subsequently, specific approaches for specific learning of syntax for code translation were proposed. These included grammar-driven approaches, tree-based approaches, and more recently, pretrained model-based approaches. Due to the lack of substantial paired training data, researchers have also applied unsupervised and semi-supervised learning approaches to code translation to mitigate the effect of the lack of large-scale paired training data. We will describe some of these in greater detail in Section 2: Literature Review.

In this work, we replicate the baseline GraphCodeBERT (Guo et al., 2021) and further evaluate it extensively on existing datasets as well as a self-curated evaluation set generated from handpicking paired code samples from Leetcode for Java and C#. We train and evaluate the model for both Java2C# and C#2Java settings. For a more thorough evaluation, we run quantitative evaluation on additional metrics as used by Ahmad et al. (2023). We extensively show the shortcomings of the model as well as the current status of the code translation research domain in general and propose our own approach for mitigating these issues to some extent without incurring a substantial fine-tuning cost or the need for a gigantic paired dataset for supervisee training. Our approach can also enable zero-shot translation for certain language pairs with reasonable performance. We elaborate on our experiments, curated test dataset from Leetcode, ongoing fine-tuning dataset curation from Codeforces, proposed approach for Assignment 4, and our intuition behind the results as well as the reasoning behind our proposed architecture in detail below.

2 Literature Review

Code translation is a specialized task in the domain of machine translation, specifically for the purpose of effectively and correctly translating code from a given programming language to another. Over the years, several works have been proposed to enable end-to-end code translation between various coding languages, enabling programmers to achieve a

[1] `https://github.com/Siddharth-Gandhi/GraphCodeBERT_translation`

variety of tasks. These can be upgrading code from deprecated frameworks or languages (Aggarwal et al., 2015), translating code from between commonly used languages like C++, Java and Python for general purpose or specific use cases, etc. Some of the earlier approaches used lexical features for code translation (Nguyen et al., 2013) or phrase-based code translation (Karaivanov et al., 2014), which was one of the original backbones of commonly used machine translating tools like Google Translate.

Later works relied on tree-based neural networks for code translation. For example, using the dataset curated by Nguyen et al. (2013), Chen et al. (2018) proposed the first neural network for code translation, using the hypothesis that code translation is a modular procedure in which a sub-tree of the source tree is translated into the corresponding target sub-tree at each step, thus employing a tree-based neural network for code translation between C# and Java. These approaches are, however, limited to a few language pairs for which small parallel datasets were created manually (e.g. C# to Java) or can be created with rule-based tools.

More recently, unsupervised machine translation-based methods were proposed for code translation. For example, Roziere et al. (2020) proposed TransCoder, to translate between Python, Java, and C++, showing that their approach significantly outperformed rule-based baselines without requiring any parallel data or expert knowledge, while being capable of generalizing to other languages. anne Lachaux et al. (2021) later introduced a deobfuscation objective along with the masked language modeling objective introduced by Devlin et al. (2019). This led to substantially improved performance over TransCoder.

Rozière et al. (2021) showed that several prior works on unsupervised code translation rely on back-translation, which is not an ideal solution, since code is highly sensitive to minor changes; even a single token can result in compilation failures or erroneous programs, unlike natural languages where small inaccuracies may not change the meaning of a sentence. They proposed an automated unit-testing system to filter out invalid back-translations for creating a fully tested and noise-free parallel corpus. Fine-tuning an unsupervised model on these cleaned samples helped their model easily outperform prior works.

Guo et al. (2021) presented a pre-trained model for code languages that considers the inherent structure of code. They use data flow, a semantic-level structure of code that encodes the relation of "where-the-value-comes-from" between variables, instead of the syntactic-level code structure like abstract syntax tree (AST), making the model more efficient. They further introduce two structure-aware pretraining tasks along with masked language modeling: predicting code structure edges and aligning representations between source code and code structure. Their model achieved state-of-the-art performance not only for code translation, but also for three other tasks: code search, clone detection, and code refinement. This is the baseline that we have chosen to implement in this assignment.

Recently, Ahmad et al. (2023) proposed the task of performing back-translation via code summarization and generation. During the former, the model learns to generate natural language summaries of the given code snippets. In the latter, the model learns to do the opposite, i.e., generate code snippets from natural language summaries. Therefore, target-to-source generation in back-translation can be viewed as a target-to-language-to-source generation. Their model performed comparably to their baselines.

3 Baseline Results and Analysis

3.1 Baseline Model and Replicating the Original Results

For our Assignment 3, we replicated GraphCodeBERT (Guo et al., 2021) as the baseline model, running several quantitative and qualitative results on the model for Java2C# and C#2Java settings. We not only use the metrics for predictive performance used in the original paper but also use additional metrics used by Ahmad et al. (2023). We used the pretrained weights made available for the pretrained model on HuggingFace[2] and further ran the fine-tuning experiments to replicate the results on the dataset provided in the paper's repository. We achieve almost identical performance as mentioned in the paper. Our replicated results along with the original results can be found in Tables 1 and 2.

The reasons why we chose this particular model as the baseline are as follows:

1. The work is considered a prominent work in the domain of code translation, having been

[2]https://huggingface.co/microsoft/graphcodebert-base

accepted at ICLR 2021. It has been cited over 520 times to date.

2. The pretrained model's weights were available on HuggingFace for use. We only needed to fine-tune the model, and not pretrain the model from scratch, which was in the computational resource limit available to us.

3. The paper introduces a pretrained model for code-specific language tasks, and further fine-tune the model on a dataset with paired Java and C# samples proposed in the works by Nguyen et al. (2015) and Chen et al. (2018). This approach is still used by most models for code translation by pretraining models in an unsupervised manner on large-scale monolingual or multilingual (in the context of code) corpora and then further fine-tuning them using a paired code translation dataset for the required setting. For example, Ahmad et al. (2023) also does something similar, but its pretrained model weights were not available to us and pretraining the model was outside the scope of our available computational resources. Thus, the approach is still relevant today.

4. The model achieved state-of-the-art performance on code translation when it was released and still ranks well among models for code-specific tasks, especially among models trained in a similar computational budget. This makes it a suitable baseline for code translation and a good indicator of the current status of code translation research and the gaps in current methods.

5. C# to Java translation saw slightly higher performance across metrics in general for several types of questions and especially for codes with functions and classes, potentially showing that it is easier for the model to learn translation in this direction than vice versa.

One minor issue that we noted in the dataset used by Guo et al. (2021) was that the *if* and *for* loops and their parenthesis sometimes had a space in between them (e.g. *"if ("*), and sometimes they didn't (e.g. *"if("*). This often led to the model predicting a space where the ground truth didn't have it, and the model not predicting a space when the ground truth had it. We removed the spaces from all data points that had space from the dataset, both from the training and the testing sets. This led to the slight increase in performance that we observe in Tables 1 and 2 compared to the results reported by Guo et al. (2021).

3.2 Additional Quantitative Experiments

A common trend we observed across a variety of code translation papers was that the datasets used to train the models had extremely basic codes, often not being modular enough to include functions or classes. We felt that these were far from a true representation of real-world code translation situations, and the complexity of code in the real-world scenarios (in terms of how convoluted and modular is the code) is much higher. Thus, for a more realistic evaluation of the model, we extracted 31 paired codes from various code challenges on LeetCode in Java and C# for some famous coding challenges. To ensure a comprehensive evaluation, we extracted LeetCode easy (9), medium (16), and hard questions (6), with (9) and without classes and functions (22), so that we could evaluate how the model performs across code translation of various degrees of difficulty, as well as codes that are hard-coded and those that are modular and include classes and functions. We chose this distribution, as it mimics the distribution of easy, medium, and hard questions on LeetCode. Our results for these experiments can be found in Tables 3, 4, 5 and 6.

3.3 Metrics for Quantitative Evaluation

We evaluate the performance of GraphCodeBERT not only on BLEU (Papineni et al., 2002) and Exact Match Accuracy (Acc/EM), as used in the original paper, but also on additional metrics used in other papers like Ahmad et al. (2023). All our quantitative experiments in Tables 1, 2, 3, 4, 5, and 6 are evaluated on all of these metrics. A description of all metrics used is as follows:

- **BLEU:** Evaluates the overlap of n-grams between generated code translation and the provided ground truth translations.
- **Exact Match Accuracy:** Indicates the percentage of generated code translations that exactly match with the ground truth translations.
- **Weighted N-gram Match:** Evaluates the overlap of n-grams between generated code translation and the provided ground truth

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
GraphCodeBERT	80.58	59.4	-	-	-	-	-
Our Replication	81.59	60.6	82.23	88.79	87.91	85.13	81.59

Table 1: Model performance during replication of the fine-tuned results for GraphCodeBERT on code translation for Java to C#.

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
GraphCodeBERT	72.64	58.8	-	-	-	-	-
Our Replication	72.49	60.1	73.23	85.79	83.11	78.64	72.49

Table 2: Model performance during replication of the fine-tuned results for GraphCodeBERT on code translation for C# to Java.

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
Combined (31)	31.44	0	33.69	60.96	40.63	41.67	31.41
LC Easy (9)	45.82	0	46.06	79.6	63.35	58.66	45.64
LC Medium (16)	26.88	0	29.59	57.63	37.86	37.98	26.83
LC Hard (6)	32.44	0	35.61	57.94	31.6	39.37	32.33

Table 3: Fine-tuned model performance of GraphCodeBERT on code translation for Java to C# on our LeetCode evaluation dataset.

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
Has Functions or Classes (9)	6.44	0	13.8	43.1	19.03	20.58	6.39
No Functions or Classes (22)	43.59	0	44.46	69.49	51.09	52.15	43.55

Table 4: Fine-tuned model performance of GraphCodeBERT on code translation for Java to C# on our LeetCode evaluation dataset.

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
Combined (31)	31.11	3.23	32.55	57.95	42.56	41.03	31.07
LC Easy (9)	50.12	11.11	50.55	76.55	73	62.51	49.92
LC Medium (16)	25.22	0	26.82	54.38	35.76	35.53	25.15
LC Hard (6)	32.68	0	34.55	54.24	40.25	40.39	32.53

Table 5: Fine-tuned model performance of GraphCodeBERT on code translation for C# to Java on our LeetCode evaluation dataset.

Method	BLEU	Acc/EM	weighted_ngram_match	syntax_match	dataflow_match	CodeBLEU	Moses BLEU
Has Functions or Classes (9)	10.16	0	16.92	44.08	22.98	23.52	10.1
No Functions or Classes (22)	41.27	4.55	42.21	64.71	52.23	50.09	41.21

Table 6: Fine-tuned model performance of GraphCodeBERT on code translation for C# to Java on our LeetCode evaluation dataset.

translations by assigning different weights or importance values to individual n-grams based on their relevance or significance (calculated using frequency-based weights or TF-IDF weights) in capturing the syntactic description of the code.

- **Syntax Match:** Assesses how well the generated code matches the syntactic structure of the target programming language by breaking down the code into its constituent elements, such as tokens, expressions, statements, and overall syntax tree, and then comparing the syntax trees by measuring the similarity between them.

- **Dataflow Match:** Approach is taken from compiler design and program analysis to gather information about the possible set of values that a variable can take during the execution of a program.

- **CodeBLEU (Ren et al., 2020):** Based on BLEU in the sense that it incorporates n-gram match from it while injecting code syntax in the form of abstract syntax trees and code semantics via data flow.

- **Moses BLEU:** We employ the implementation from Moses[3] open-source machine translation package. It computes BLEU score with multiple references, hence is referred to as Multi BLEU too.

3.4 Analysis of Quantitative Results Obtained

We evaluated the translation generated in both Java2C# and C#2Java settings by the model using our curated Leetcode translation evaluation set and observed the following:

1. LeetCode easy questions in the evaluation dataset achieved significantly higher evaluation metric scores than LeetCode medium and tough code translations, potentially due to the fact that they typically contained lesser use of functions where the models (both Java2C# and C#2Java) faltered more.

2. LeetCode medium questions achieved the lowest score out of all the LeetCode question types, potentially due to the fact that their solutions were longer and had the most modular solutions.

3. Very few translations were able to achieve exact match results as the ground truth on the LeetCode test set.

4. Samples with functions or classes achieved much lower performance than samples without functions or classes, showing an inherent deficiency of code translation models in handling modular codes.

3.5 Qualitative Results and Their Analysis

We also extensively analysed the performance of the models on the LeetCode dataset that we created qualitatively for each sample in both settings: Java2C# and C#2Java. Some of the notable results that highlight the model's inability to perform well for specific types of problems and the specific issues that arise in the translated code in those examples are shown in Table 7.

[3] https://github.com/moses-smt/mosesdecoder

Some key general trends in the issues in the translated code that we observed during the qualitative analysis are as follows:

1. Reasonable performance (although far from error-free translation) can be achieved typically when the model has one or fewer functions.

2. The model commonly made errors with capitalization of variable names, parenthesis, syntax errors, incomplete or merged functions, renaming variables, and function names, among other less commonly observed issues.

3. The model doesn't translate classes taken as input correctly, which is a critical use case as regular programs are typically written using classes.

4. The models perform poorly when code contains multiple functions, often not translating one function entirely.

5. Recursive functions can't really be tested, as they are not handled by the model.

6. In Java2C# setting, the model likes to override a lot of methods even when the original Java code doesn't override.

7. The model seems to be overfitted to the training data and the val/test sets used in their experimental setup are not a true representation of real code generally. For example, input is only functions but in real-life code, we often input file.java and need a file.cs output.

Our entire evaluation of the 31 examples of LeetCode translations along with basic descriptions of the errors obtained can be found here[4] in the sheets java_analysis and cs_analysis.

Another critical thing to note is that we ran the translated codes generated in both Java2C# and C#2Java settings and out of all 31 examples for each, only 1 code per setting was able to run without errors and give correct results. These can be found in the result sheets linked above. The samples for each setting that run correctly are different.

[4] https://docs.google.com/spreadsheets/d/1IQHH7A3d6IZ3SKCBDOBiFXZNog6XDwGN1vzUmO9SRMg/edit#gid=2011051207

Problem	Ground Truth Code	Translated Code	Reasons for Failure

Easy: Linked List Cycle	`public boolean hasCycle(ListNode head) {`
	`  ListNode slow = head, fast = head;`
	
	`  while (fast != null && fast.next != null) {`
	`    slow = slow.next;`
	`    fast = fast.next.next;`
	
	`    if (slow == fast)`
	`        return true;`
	`  }`
	
	`  return false;`
	`}`
		`public boolean hasCycle(ListNode head){`
		`      ListNode Slow = head, fast = head;`
		`      while(fast != null && fast.next != null){`
		`        fast = fast.next;`
		`        fast = fast.next.next;`
		`        if(fast == fast) return true;`
		`        return false;`
		`        }`
		`}`
			1. Renames slow to fast
			2. Missing closing parentheses for main function
			3. Variable slow renamed to Slow

Medium: Snakes and Ladders	`public int SnakesAndLadders(int[][] board) {`
	`    int n = board.Length;`
	`    Queue<int> queue = new Queue<int>();`
	`    queue.Enqueue(1);`
	`    bool[] visited = new bool[n * n + 1];`
	`    for (int move = 0; !queue.Count.Equals(0); move++) {`
	`        for (int size = queue.Count; size > 0; size--) {`
	`            int num = queue.Dequeue();`
	`            if (visited[num]) continue;`
	`            visited[num] = true;`
	`            if (num == n * n) return move;`
	`            for (int i = 1; i <= 6 && num + i <= n * n; i++) {`
	`                int next = num + i;`
	`                int value = GetBoardValue(board, next);`
	`                if (value > 0) next = value;`
	`                if (!visited[next]) queue.Enqueue(next);`
	`            }`
	`        }`
	`    }`
	`    return -1;`
	`}`
	
	`private int GetBoardValue(int[][] board, int num) {`
	`    int n = board.Length;`
	`    int r = (num - 1) / n;`
	`    int x = n - 1 - r;`
	`    int y = r % 2 == 0 ? num - 1 - r * n : n + r * n - num;`
	`    return board[x][y];`
	`}`
		`public virtual int ParseAndLayers(int[][][] tab){`
		`        int n = section.Length;Queue<int> queue = new List<int>();`
		`        bool[] visited = new List<int>();`
		`        bool[] visited = new bool[n * n + 1];`
		`        for(int move = 0;!queue.Count;move++){`
		`            for(int size = queue.Count;size > 0;size--){`
		`                for(int size = q.Count;size > 0;size--){`
		`                    int num = true;`
		`                    if(num == n * n){`
		`                        return move;`
		`                    }`
		`                    for(int i = 1;i <= 6 && num + i <= n;i++){`
		`                        int next = num + i;`
		`                        int value = GetBoardValue(board, next);`
		`                        if(value > 0){`
		`                            next = value;`
		`                        }`
		`                        if(!visited[next]){`
		`                            queue.Offsets(next);`
		`                        }`
		`                    }`
		`                }`
		`            }`
		`        return -1;`
		`    }`
		
		`private int GetBoardValue(){...}`
			1. Unable to take in multi-function or multi-class inputs directly. It only processes one function at a time
			2. Makes the function virtual which was not originally the case (overfitting to train data probably)
			3. Missing closing } parentheses
			4. NameError: renames original function (from Java) snakesAndLadders to ParseAndLayers.

Hard: IPO	`public int findMaximizedCapital(int k, int W, int[] Profits, int[] Capital) {`
	`        PriorityQueue<int[]> pqCap = new PriorityQueue<>((a, b) -> (a[0] - b[0]));`
	`        PriorityQueue<int[]> pqPro = new PriorityQueue<>((a, b) -> (b[1] - a[1]));`
	
	`        for (int i = 0; i < Profits.length; i++) {`
	`            pqCap.add(new int[] {Capital[i], Profits[i]});`
	`        }`
	
	`        for (int i = 0; i < k; i++) {`
	`            while (!pqCap.isEmpty() && pqCap.peek()[0] <= W) {`
	`                pqPro.add(pqCap.poll());`
	`            }`
	
	`            if (pqPro.isEmpty()) break;`
	
	`           W += pqPro.poll()[1];`
	`        }`
	
	`        return W;`
	`    }`
		`public int findMaximizedCapital(int k, int W, int[] Profits, int[] Capital){`
		`        List<Integer> projects = new ArrayList<>();`
		`        for(int i = 0;i < Profits.length;i++){`
		`            pProjections.add((short[i],Profits[i]));`
		`        }`
		`        pProjects = positions.toArray(pow[i], Profits[i]));`
		`        List<Integer> availableProjects = new ArrayList<>`
		`        (p.sizeByPrefix).toList();`
		`        int index = 0;`
		`        for(int i = 0;i < k;i++){`
		`            while(index < commits.size() && does[index].get Capital();}`
		`            if(!availableProjects.isEmpty())break;]`
		`            maxProfitProjects.add(p);`
		`            }`
		`            return W;`
		`        }`
			1. Plain incomplete and bad code
			2. Many syntax errors
			3.Uses List instead of priority queue:
			Importantly, PriorityQueue in C# was only available in .NET 6, hence why we use List in the gold standard.
			But the model has no way of knowing
			this versioning problem which can potentially lead
			to using different data structures (specific to language)
			leading to bugs or slowdown s
			4. NameError: renames original function (from Java)
			snakesAndLadders to ParseAndLayers.

Table 7: Qualitative Analysis across certain examples from our LeetCode evaluation dataset. The first and third examples are from C#2Java while the second example is from Java2C#.

3.6 Other Drawbacks of GraphCodeBERT

To conclude, some of the drawbacks of GraphCodeBERT are as follows:

1. Performs well on the provided val/test sets but does not generalize to more realistic code for translation.

2. Fairly computationally expensive to fine-tune, and the entire 172M parameters are trained during fine-tuning. While that is not exorbitant when compared to some extremely large language models available nowadays, it is still not an efficient process and requires a fair amount of compute.

3. Relies entirely on available paired data for translation and cannot be used to do zero-shot translation to other languages.

4. Not modular to swap certain components from the model to improve performance.

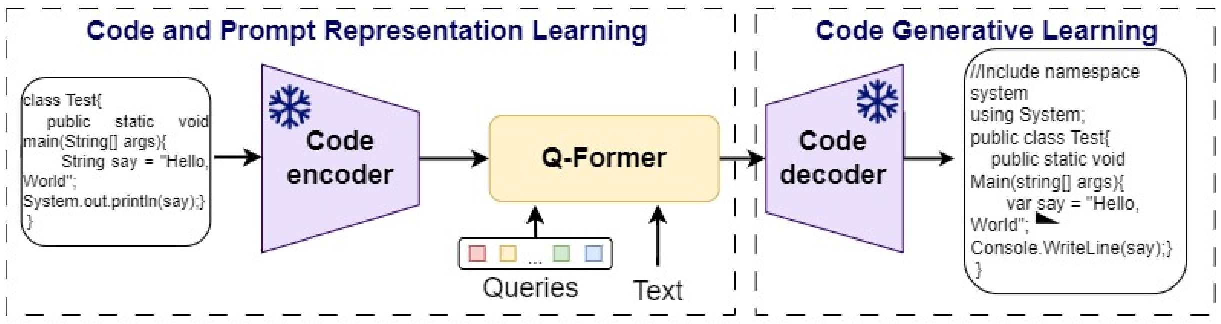
5. Despite being a contextual model, its understanding of code syntax is still limited, as we can see it frequently makes syntax errors with parenthesis and other language-specific syntactical rules.

4 Our Proposed Approach

In this section, we propose our model for code translation that tries to mitigate the various issues we observed with the baseline model GraphCodeBERT in the above sections.

Our model is based on the architecture of the multi-modal image-to-text model BLIP-2 (Li et al., 2023), as it uses frozen encoder and decoder models pre-trained on code data. Unlike BLIP-2, our model, which we intend to call **CodeBLIP**, has unimodal frozen code transformer models, which will be taken off-the-shelf from available open-source code languages available on HuggingFace. We intend to align these for code translation (and in the future, extend to other tasks like code refinement) using an intermediate transformer model called the Querying-Transformer or the Q-former. The Q-former is the only component of the model whose parameters are trained during fine-tuning, and since we intend to use openly available code encoders and decoders in our model, we do not need to pretrain our own models. This was an issue that we faced with certain recent baseline papers like Ahmad et al. (2023), whose pretrained model weights weren't available. Further, the total trainable parameters of the model are expected to be in the range of 80M, which is significantly lower than baselines like Guo et al. (2021) and Ahmad et al. (2023), which have 172M and 140M trainable parameters respectively.

The model is learned in two stages: representation learning to align the code encoder with the Q-former and generative learning for aligning the



```
class Test{
   public static void
main(String[] args){
      String say = "Hello,
World";
System.out.println(say);}
}
```

```
//Include namespace
system
using System;
public class Test{
   public static void
Main(string[] args){
      var say = "Hello,
World";
Console.WriteLine(say);}
}
```

Figure 1: Graphical representation of our proposed model for code translation using two frozen pretrained transformers aligned using a Q-former. This figure shows the training paradigm we intend to use in our model for mitigating some of the issues seen with the baseline and prior works.

Q-former with the text decoder for code generation. This is similar to the training paradigm used by BLIP-2. Our training methodology can be pictorially represented as shown in Figure 1. The Q-former is trained on three objectives: code matching, code generation, and contrastive learning.

The encoder and decoder in our model will be reasonably new multilingual code transformer models with weights openly available so we can use them without having to pretrain encoders ourselves. Aligning these models will ensure that the representations between the input and the output are matched accordingly and the model can fine-tune better on the required downstream task and performs. Further, once the model representations are aligned using the Q-former, we expect this to enable downstream translation into languages other than the ones that were used for supervised training for code translation, as long as the model was learned on the language during pretraining, thus enabling zero-shot translation.

Since only the Q-former is trained in the model, we can swap the encoder and the decoder as better contextual models are introduced in the future. The modular approach and the relatively lower expected computational cost of CodeBLIP will ensure that it will be able to incorporate the advantages of newer models without the need to start from scratch. Advancements to either the encoder, the decoder or both can be incorporated individually.

We initially intend to train the Q-former using existing paired datasets for code, starting with Java and C# paired data. For this, we are scraping code from codeforces from various coding challenges for different coding languages and pairing them. We will filter the pairs using existing code translation models to ensure that noisy samples are not included since we curate since the codes extracted from codeforces can be from different users with different potential solutions being introduced.

In the future, we will also be training the model for other language pairs to make the model more robust for translation, as well as for data for other code-related tasks like code refinement within the same language, if the same language is supported by the encoder and the decoder, thus making the model more general purpose.

4.1 Expected Advantages of Our Proposed Approach

To summarise, some of the advantages we expect to see from our proposed model to mitigate the issues with the baseline models are as follows:

1. Ability to use various large frozen transformer models, thus having the ability to take as input and process longer sequences of code as input for translation.

2. Potentially enable zero-shot translation to languages seen only when the transformer was pretrained but not during any fine-tuning. This mitigates the requirement of necessary paired data for each language pair for translation.

3. Lower fine-tuning cost than other baselines, since only the Q-former's parameters are being trained, while all others are frozen.

4. Likely to be more robust to noise in the training dataset.

5. Improved performance over previous baselines for code translation.

6. Ability to perform other tasks like code refinement in the future with minimal fine-tuning.

References

Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. Using machine translation for converting python 2 to python 3 code. *PeerJ Prepr.*, 3:e1459.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and generate to back-translate: Unsupervised translation of programming languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1528–1542, Dubrovnik, Croatia. Association for Computational Linguistics.

Marie anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. In *Advances in Neural Information Processing Systems*.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA. Curran Associates Inc.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, page 173–184, New York, NY, USA. Association for Computing Machinery.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA. Association for Computing Machinery.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA. Curran Associates Inc.

Baptiste Rozière, J Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *ArXiv*, abs/2110.06773.