Soliman *et al.*
Journal of Engineering and Applied Science (2022) 69:104
https://doi.org/10.1186/s44147-022-00159-4

Journal of Engineering and Applied Science

RESEARCH Open Access

MarianCG: a code generation transformer model inspired by machine translation

Ahmed S. Soliman[1,2*], Mayada M. Hadhoud[1] and Samir I. Shaheen[1]

*Correspondence: ahmed.shokry@eng1.cu.edu.eg

[1] Department of Computer Engineering, Cairo University, Giza, Egypt
[2] Department of Computer Engineering, Al-Azhar University, Nasr City, Egypt

Abstract

The idea that computers can build their own programs is extremely significant, and many researchers are working on this challenge. Code generation is described as the process of generating executable code that can be run directly on the computer and fulfills the natural language requirements. It is an intriguing topic that might assist developers to learn a new software technology or programming language, or it could be a simple technique to help in coding through the description of the natural language code developer. In this paper, we present MarianCG, a code generation Transformer model used to tackle the code generation challenge of generating python code from natural language descriptions. Marian neural machine translation (NMT), which is the core model of the Microsoft Translator, is the basis for our NL-to-Code translation engine and is the heart of the teaching model. MarianMT is the teacher language model in our study, and it is one of the most successful machine translation transformers. In our approach, we use a sinusoidal positional embedding technique to represent the position of each token in the text, as well as no layer normalization embedding. Our code generation approach, MarianCG, is based on fine-tuning a machine translation pre-trained language model. This allows us to demonstrate that the pre-trained translation model can also operate and work as a code generation model. The proposed model outperforms recent state-of-the-art models in the problem of code generation when trained on the CoNaLa and DJANGO datasets. MarianCG model scores a BLEU score of 34.43 and an exact match accuracy of 10.2% on the CoNaLa dataset. Also, this model records a BLEU score of 90.41 and an exact match accuracy of 81.83% on the DJANGO dataset. The implementation of MarianCG model and relevant resources are available at https://www.github.com/AhmedSSoliman/MarianCG-NL-to-Code.

Keywords: Code generation, Natural language programming, MarianCG, CoNaLa, MarianMT, Marian NMT, Neural machine translation

Introduction

Code generation is a significant field that can predict and generate suitable code as output from the natural language as the input source. The increasing of code generation tools with accuracy and optimization tools can help to increase the productivity of the programming tools [1]. Application Programming Interfaces or APIs make software development and innovation easier by allowing applications to share data

and functions in a simple and safe manner. An API is a set of computer instructions and procedures that may be used to get access to a website or web-based software application. Automatic code generation might help developers learn a new programming language or deal with new APIs.

Nowadays, pre-trained language models witnessed tremendous success in the NLP field [2]. A pre-trained model is a model that has been trained on a big benchmark dataset to tackle some problem and then save this network with weights to be trained and reused for another task. Pre-trained models are commonly used to be the core of the transfer learning job. Through pre-training and fine-tuning, we can enhance model robustness and uncertainty. There are several approaches that enable pre-trained language models to train massive models with billions of parameters from large-scale unlabeled corpora in a self-supervised manner. Recent researches [2–7] have shown using pre-trained models and also demonstrated the benefits of employing pre-trained language models for many tasks such as question answering, text classification and machine translation.

Transformers contain numerous pre-trained models that can be used for a variety of tasks and datasets [8]. Transformers have demonstrated that they can both be few-shot [9] and unsupervised multitask [10] learners. Transformers prove that they can be applied to any pipeline tasks like machine translation, text-to-text generation, classification, and other tasks. Furthermore, researchers demonstrated that massive pre-trained language models can be few-shot semantic parsers [11].

Contributors can use the Transformers library to publish language datasets and generate and distribute pre-trained models to get new models with high performance and huge results. In May 2020, the University of Helsinki's Language Technology Research Group (Helsinki-NLP) developed and submitted a huge set of translation models to the Transformers library called MarianMT [12]. They constructed their models depending on Marian [13] neural machine translation (MarianNMT) framework which is accessible at https://www.marian-nmt.github.io, and it is published under the MIT license. The MarianNMT framework and the Open Parallel Corpus (OPUS) dataset were used to train the Helsinki-NLP machine translation models to get MarianMT model.

With limited computing resources, it is possible to train translation models that are competitive with state-of-the-art models. Adapting a pre-trained language model with the same architecture from one task to another is a crucial stage in generating a new trustworthy, reliable, and effective model.

We implemented MarianCG which is a Transformer language model that can work in the code generation task. This is accomplished by fine-tuning MarianMT which is a pre-trained language model with CoNaLa [14] and DJANGO [15] datasets. MarianCG model is shown in Fig. 1. We applied the BLEU score measure [16] and exact match accuracy to solve the code generation problem, which other researchers used to quantify the quality of the generated output.

The experimental findings on the CoNaLa and DJANGO datasets reveal that the MarianCG transformer model outperforms other state-of-the-art models in respect of the relevant evaluation criteria.

Our main contributions are:

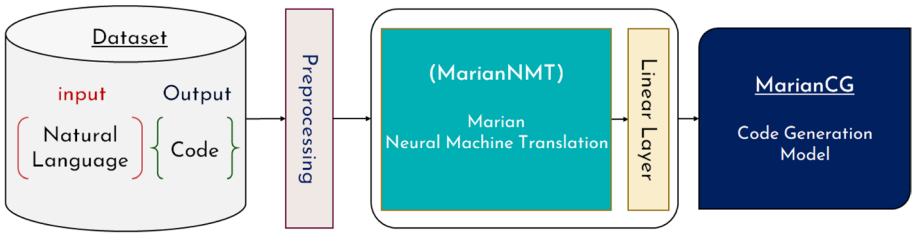


Fig. 1 MarianCG model for code generation

1 Introducing MarianCG transformer model, which is a code generation model capable of creating code from natural language
2 Testing the effectiveness of using Marian machine translation model for solving the problem of code generation
3 Demonstrate that a machine translation model may be used as a code generation model
4 Setting the new code generation challenge contributors, with a BLEU score of 34.43 and 10.2% exact match accuracy on the CoNaLa dataset. Also, we recorded the highest accurate results on the DJANGO challenge reaching 81.83% exact match accuracy, and a BLEU score of 90.41

The rest of the paper is organized as follows: Section 2 summarizes the relevant related work and discusses the previous techniques to solve the code generation task. Also, it sets this work apart from the relevant related work. Section 3 provides a description of the core model, Marian, and what inspired us to use MarianMT transformer machine translation model in the code generation problem. Section 4 provides an overview of the proposed model and its components. Section 5 contains a list of the datasets that we use in our experiments. Following that is a section covering implementation and experimental work, which includes the evaluation metrics and experimental setup. We gain results compared to other researchers through the studies after the implementation section then the discussion section that discusses our work. Finally, the section that concludes the paper to demonstrates how our technique adds value to the code generation task and the future work of our study.

Related work

The problem of transforming natural language (NL) descriptions to generate executable code is known as code generation, which is a sub-task of semantic parsing. There are some difficulties in this problem because the output has a well-defined structure and the domain, structure of the input, and output are not similar. Techniques that are used for solving this problem can be divided into tree-based techniques and deep learning based techniques.

Tree-based techniques

Tree-based techniques are considered one of the task-driven forms of semantic parsing that translate the natural language input to formal machine executable representation. These techniques can represent code as abstract syntax trees (ASTs) which can be

described as the syntactic tree representation of the target code or the cleaned-up version of the parse tree that captures the structure of expressions, the program's control components.

The goal of the ASTs is simply to describe the semantic structure of sentences in a computer language as trees. Semantics can be stated with attribute grammar, but most semantic approaches are significantly more intelligible when based on a clearer representation of a language's phrases. There are standard templates for the various components of a programming language definition when simulating the code as AST. Also, keep in mind to define the code as AST you need to know the collection of syntactic categories or domains and a set of rules to describe how to connect these categories with each other.

Code generation and semantic parsing need to convert unstructured (or partially structured) inputs to well-formed, executable outputs. So, researchers have used sequence-to-tree models for code generation, with the tree representing the AST of the target source code [11, 17–25], because they wanted to improve the process of creating code snippets by the ASTs.

Advantages and disadvantages of tree-based techniques

There are several benefits to implementing tree-based approaches in this task, such as handling the code generation problem by converting the natural language input to the matching AST, which can assist improve accuracy by requiring the output code to be represented with a well-formed structure. Furthermore, tree-based techniques may be used to any type of data and can also manage data that is not generally distributed. Furthermore, tree-based techniques are easy to visualize, making a complex predictive model much easier to understand. Finally, because variable transformations are unnecessary, tree-based techniques need the minimum amount of data preprocessing.

On the other hand, there are some lacks for using these techniques because describing code as AST is difficult way because the number of nodes in the tree frequently surpasses the length of the natural language description. For this reason, tree-based techniques are not frequently able to produce correct code for the related natural language description which is uncommon in the training data. Also, generating AST is synchronous (the output structure diverges from the input structure). The use of ASTs has achieved less accurate results compared to deep learning-based models. There has been relatively less work on utilizing the parse trees of the natural language input. Because of these reasons, researchers turned their direction to deep learning based techniques, where there is no need to construct a tree to generate code.

Deep learning-based techniques

Source code generation is considered as text-to-text or sequence-to-sequence, which can be developed and maintained by deep learning models. Machine intelligence that understands and creates the complex structures of software has a lot of applications in software engineering. There are some sequence-to-sequence models, and these models can convert the target code into other sequence domains.

Using deep learning to solve and deal with many problems has become an important technology in various domains; therefore, numerous research projects are focused

on deep learning technology and pre-training models. Additionally, transfer learning proved great results to generate new models depending on another pre-trained model. Transfer learning is the process of fine-tuning a model that has been trained to execute one job to perform on another task. A pre-trained model can be defined as a stored network that has already been trained on a large dataset, typically on a large-scale task.

As a result, recent researchers [24, 26–31] in the code generation problem focused on fine-tuning and training the pre-trained model in order to create a new task-oriented model. The amazing potential for using transfer learning to adapt the pre-trained model to a specific job further provide consistent outcomes and findings for the seq2seq code generation task.

Previous contributors' work

In 2016, Dong and Lapata proposed a methodology for learning from natural language descriptions and meaning representations [17]. They used recurrent neural networks (RNNs) with long short-term memory (LSTM) units to encode phrases and decode logical structures for considering the task of semantic parsing. They created a technique that is based on an attention enhanced encoder-decoder model, and this technique can convert input utterances into vector representations and produce their logical forms. This is done by conditioning output sequences or trees on the vector representations. These encoded and decoded input utterances and their logical structures, and the attention layer is used to directly control the program synthesis process. Their testing results revealed that adding a hierarchical tree decoder and the attention mechanism to the model enhanced performance across the board.

In 2017, Yin and Neubig proposed a syntax-driven neural code generation technique [18] that constructs an abstract syntax tree by progressively applying actions from a grammar model. They designed a probabilistic grammar model for AST generation. The Python abstract grammar has a set of production rules, and an AST was created by combining numerous production rules, each of which consists of a head node and multiple child nodes.

In 2018, Yin and Neubig proposed TRANX [20] which parses the utterance into a formal meaning representation. TRANX was built through a transition system, and it uses this transition system to convert a natural language utterance into an abstract syntax tree (AST) through a series of tree construction actions given an input natural language utterance. The parser is then used to turn the intermediate AST into a domain-specific meaning representation, bringing the parsing process to a close. TRANX scores each hypothesis AST using a probabilistic model specified by a neural network. But the neural semantic parser, TRANX indicated an obvious issue of incoherence in generation and got results with the CoNaLa dataset as 24.30 for the BLEU score metric. Also, TRANX got accuracy of 73.7% for the DJANGO dataset.

In 2019, Yin and Neubig proposed the Reranking model [21]. They used the previous TRANX semantic parser to get the meaning representation of the input natural language as an abstract syntax tree. They added a reranking method to output the most suitable meaning representation. The reranking model is presented as a fast-iterating method to enhance the accuracy of parsing and rerank the n-best list of the representation of meaning. This can be done by using characteristics designed to

address problems in baseline models. This model is used and get results with four datasets GEO, ATIS, DJANGO, and CoNaLa. The result obtained is 30.11 of BLEU score with the developing and testing with the CoNaLa dataset. Also, the results on the DJANGO dataset were recorded with 80.2% accuracy.

In 2019, Shin et al. introduced PATOIS [22] which is a program synthesizer and also a neural program synthesizer that trains a tree-based neural synthesizer to use the code idioms while coding generation. The PATOIS system was built on top of structural generative models like graph neural networks and sequence-to-tree models.

In 2020, Xu et al. proposed a deep learning model by data re-sampling, fine-tuning the pre-trained model, and using incorporating external knowledge [24] to predict executable python code. To include external knowledge in code generation models, they suggested a model-agnostic strategy based on data augmentation, retrieval, and data re-sampling, which obtained new results on the CoNaLa open-domain code generation task. They used the CoNaLa-Mined [14] dataset, which was automatically mined from StackOverflow and contained 600,000 NL-code pairs in Python. They sorted all pairings by confidence scores and discovered that the top 100K examples have a good level of code accuracy and NL-code correlation. As a result, the top 100K couples are chosen for the tests. They generated roughly 13K different NL-code pairings (without resampling) from Python API documentation after pre-processing. They also sampled the same number of pairings for the re-sampling setting to provide a fair comparison. They used the NL-to-code generation model TRANX [20] as the basic model, with hypothesis reranking [21]. They also used length normalization [32] to make sure that beam search didn't favor shorter results over longer ones. They got 30.69 BLEU score with external knowledge with the API model, and when they added reranking to external knowledge with API they got 32.26 BLEU score metric.

In 2021, Dahal et al. proposed a paper [25] which describes the analysis of Tree-structured architecture and their effect on the code generation problem. They ran and tested text-to-tree, structured tree-to-tree, and linearized tree-to-tree models on constituency-based parse trees where their goal was generating the corresponding ASTs of the code. They used CoNaLa and ATIS datasets. Constituency or dependency trees are describing the syntactic structure of the input, and these trees can be used to accomplish subtree alignment with the destination code matching the AST and benefiting the downstream job. Their tree-to-tree model achieved good results.

In 2021, Orlanski and Gittens worked on expanding the original CoNaLa dataset to include the multimodal textual question bodies and thus the pertinent contextual information they contain such as inputs, outputs, and required libraries [27]. They developed a new transformer model trained in the BART [33] encoder-decoder model. For both the annotated and mined cases in the CoNaLa corpus, they obtained these textual bodies from Stackoverflow. Then, they used the question bodies and concatenated intents as inputs for a huge pre-trained language model and then used beam search to construct the answer code snippet. They used Python and HuggingFace's transformer package to build their model. Finally, for text generation, they employed a BART model with a linear layer and a distinct bias. They got a 26.24 BLEU score when using the BART base model and when they used the BART model with mined data, they got a 30.55 BLEU score.

In 2021, Norouzi et al. showed that transformer-based seq2seq models can compete with or outperform models created expressly for code generation [26]. They created a seq2seq transformer model, and they built this model by fine-tuning the pre-trained transformer BERT model as an encoder the decoder was the original transformer decoder with 4-layers transformer decoder. The key is to create a new model and combine the relatively large monolingual corpora of the meaning representation with traditional large-scale pre-trained encoders. They got the highest BLEU score with CoNaLa dataset that reached 33.41. Also, the accuracy scored on the DJANGO dataset was 81.03%. A Seq2Seq model transformer with a little specialized prior could potentially achieve results superior to or competitive with models specially developed for code generation and semantic parsing by leveraging a sufficiently large monolingual corpus of the programming language.

In 2022, Beau and Crabbé developed a new code generation model which has the encoder-decoder architecture [28]. They used BERT as an encoder and decoder as a grammar-based. This is some change in TranX [20] seq2seq architecture for generating code from the natural language description. Their proposed architecture can obtain an abstract syntax tree (AST) is constrained by the programming language grammar. They trained and tested their model on the CoNaLa and DJANGO datasets. This transition system is open to guarantee the generation of syntactically correct code. Their research emphasizes the significance of grammatical limitations as well as particular techniques for managing variables, list naming, and typing. They scored a 34.2 BLEU score on the CoNaLa dataset, and an accuracy of 81.03% on the DJANGO benchmark.

Marian and inspiration for code generation

Marian NMT and MarianMT

Marian is the core engine for the Microsoft Translator Neural Machine Translation services. Marian is a self-contained, free open source, and efficient neural machine translation framework which is a built-in automated differentiation engine based on dynamic computation graphs. This framework was entirely developed in C++, and it demonstrated a research-friendly toolkit with high training and translation speeds. Training Marian was performed on raw texts, with data processing employing the SentencePiece. It is being employed in several projects and is the primary translation and training engine as well as it is used by a wide range of enterprises, organizations, and research groups.

Marian holds its own position in the developing ecosystem of open-source NMT toolkits, and it has powerful translation features, best defined by these features:

1 It is self-contained, having its own back end that does reverse-mode automated differentiation using dynamic graphs.
2 It supports working on single GPU/CPU and multi GPUs/CPUs. It provides GPU/CPU translation style as well as quick multi-GPU training. Also, this model contains the feature of batch translation.
3 Marian has the feature of creating word alignments and attention output with the ability to rescore the n-best lists and parallel files.

Marian was used by the University of Helsinki's Natural Language Processing lab to train hundreds of translation models using parallel data acquired from Opus, and those models were later open-sourced and called MarianMT. Researchers in this lab subsequently adapted the trained model into Huggingface Transformers and made them accessible through the Huggingface Hub. Each MarianMT model is a transformer encoder-decoder with six layers. MarianCG was inspired by Marian machine translation, which served as the base for our code generation challenge.

Inspired by Marian transformer models for machine translation

Marian was chosen as the backbone for our code generation approach for many reasons. To begin, we can talk about the importance of pre-training transformer models and the added value of this methodology in solving many problems. Pre-training transformer models are extensively employed nowadays for a variety of tasks [34], and it has been applied in code generation in recent years with great impacts. Pre-training is also beneficial for machine translation and code generation activities. For example, the Facebook BART model is a machine translation transformer model, and it was employed and fine-tuned for the code generation problem using CoNaLa, yielding a BLEU score of 26.24. Also, the pre-trained BERT model was merged with the Transformer decoder and scored 32.57 BLEU for the code generation challenge. This motivated our new effort to use the pre-trained model and fine-tune this model to get a significant improvement for the code generation challenge. In addition, it is a simple technique to fine-tune the pre-trained model and start training the model from its final weight rather than the original weights. Furthermore, we found that there are many machine translation models with huge architecture, such as T5. To be considered these huge models to deploy, powerful resources are needed with a strong GPU and plenty of memory with high processing capability.

Marian was chosen since it is a quick neural machine translation service with accurate machine translation outputs for several languages. MarianMT's creators, Helsinki-NLP, have over 1000 pre-trained language models for MarianMT translation models. We have a vision of using a pre-trained model from one language to work in another area. This includes setting up and building AraT5 from T5 to work in another language or domain. Marian models are smaller than many other translation models in the machine translation model collection, making them perfect for fine-tuning experimentation and integration testing. Marian NMT, the core of Microsoft translator, serves as the primary fully basic model for our training to execute the MarianCG model. As a result of these observations and insights, we developed MarianCG, a code generation model influenced by the Machine Translation Transformer approach.

Proposed code generation model

MarianCG model is built and developed using Marian neural machine translation (MarianNMT). We fine-tuned MarianMT transformer which is a pre-trained model from Helsinki-NLP and got our model, MarianCG. It is a multi-head attention transformer with zero-shot learning which observes samples that were not shown during training and predicts the sentences that are the right outputs. MarianCG model got high and accurate results for the code generation problem with fast performance.

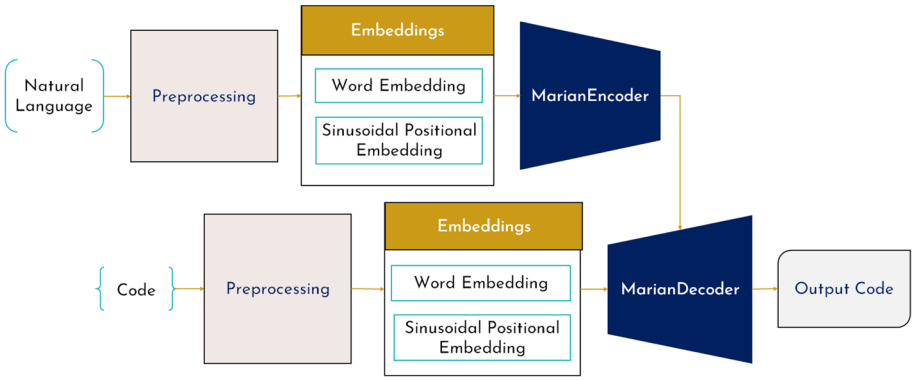


Fig. 2 MarianCG model architecture

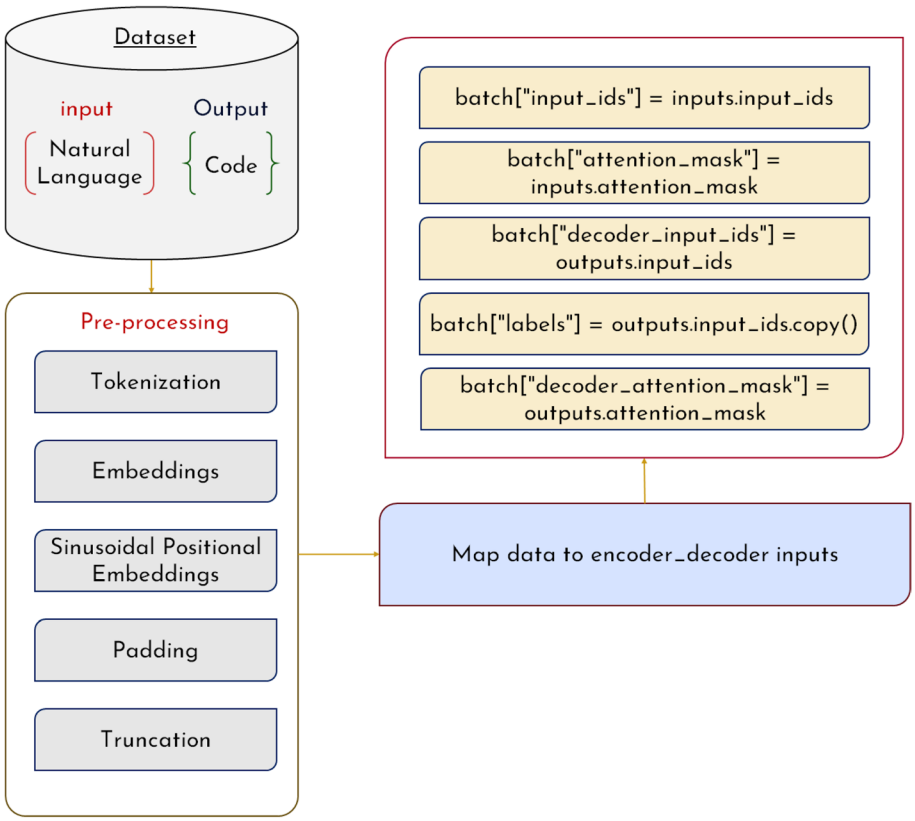


Fig. 3 Preprocessing phase for the data

MarianCG model architecture

MarianCG is trained and fine-tuned on MarianMT model that was built using Marian-NMT. MarianNMT allows rapid training and translation. The architecture of the code generation model, MarianCG is shown in Fig. 2. It starts by loading the dataset, then the preprocessing phase of the input and the output as shown in Fig. 3. Preprocessing contains tokenization of the sentence, then get embeddings of each token and do positional embedding for each token to learn each position of all tokens concerning to the specific

token. In addition, padding and truncation are part of the preprocessing phase. Finally, the input and the output sentences are directly inserted into the encoder-decoder model. MarianCG model consists of a stack of 6 layers in the encoder and a stack of 6 layers in the decoder.

MarianCG model is similar to BartForConditionalGeneration with a few minor modifications:

- Static (sinusoidal) positional embeddings.
- No layer normalization embedding.
- The first token in the generation task is the pad token which has 0 as a token embedding.

After the attention softmax, the encoder's attention weights are used to compute the weighted average in the self-attention heads. MarianCG is a PyTorch model with coding and implementation of the Marian neural machine translation transformer.

Figure 4 shows the two representations of the same example of code generation. The first representation in Fig. 4a is how to find a good solution for code generation using the representation of manual abstract syntax tree (AST). The second representation in Fig. 4b shows the same example with MarianCG encoder decoder transformer model that successfully gets highly accurate results compared to AST.

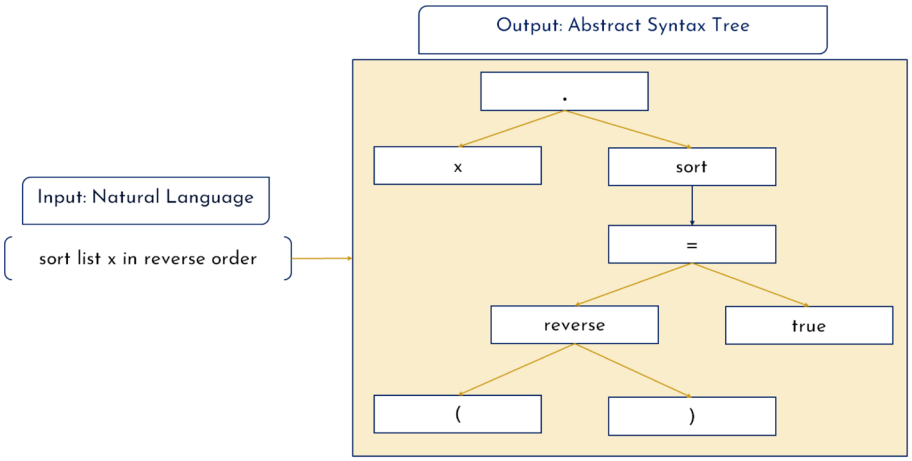
MarianCG tokenization

Marian tokenizer is developed and mainly depends on SentencePiece [35]. SentencePiece is a text generation neural network with a text tokenizer and detokenizer which has the predetermined prior vocabulary size to the training of the neural model. SentencePiece extends direct training from raw sentences to implement subword units like unigram language model [36] and byte-pair-encoding (BPE) [37]. Marian tokenizer is derived from PreTrainedTokenizer in the huggingface transformer library, which includes the majority of the essential methods.

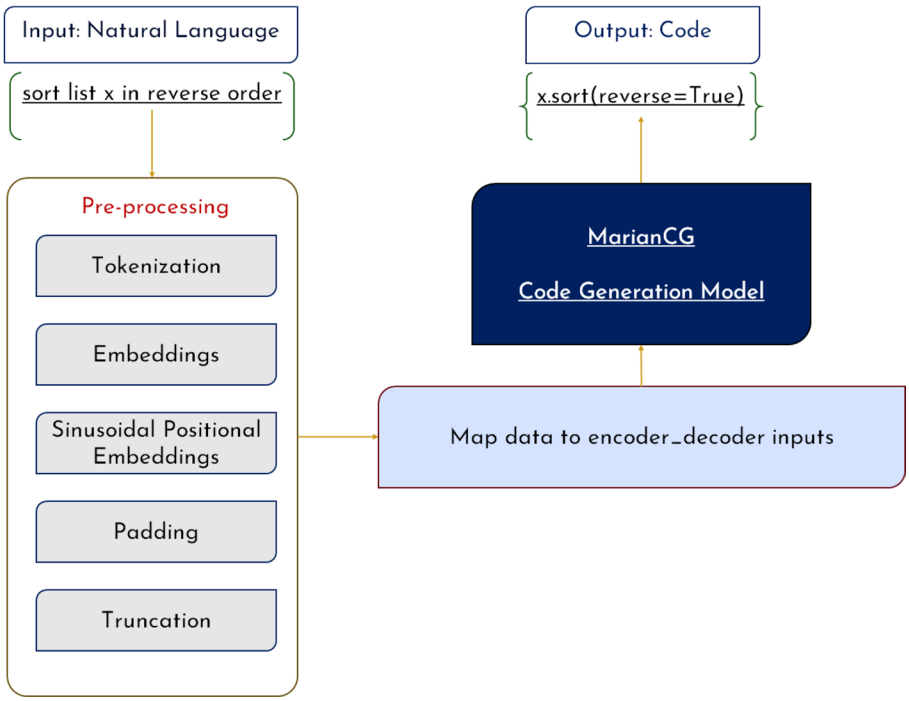
MarianCG embedding and sinusoidal positional embedding

MarianCG model does not contain convolution or recurrent neural networks, so the role of the embedding and positional embedding now is important and clear. So, by determining data about the relative or absolute location of the tokens in the sequence to have the order of the sequence. The positional and word embeddings are shared between the encoder and decoder. MarianCG model contains sinusoidal positional embeddings to the input embeddings at the encoder and decoder. The job of positional embedding is to provide information about the location of each token. This enables the attention layer to compute context-dependent responses, such that two tokens with the same value in the input phrase receive distinct representations.

Transformers employ sinusoidal positional encoding to represent the position of the input. Sinusoidal positional embedding calculates the position encoding as a mix of sine and cosine functions with geometrically increasing wavelengths. The sinusoidal representation works as well as a learned representation and better generalizes sequences that are longer than the training sequences.



(a) Abstract Syntax Tree representation for the Code Generation problem



(b) MarianCG Encoder Decoder representation for Code Generation

Fig. 4 Example of code generation

Positional encoding is defined and formulized in paper [38] where the sum of positional encoding and token embedding is given to the encoder and decoder input layers of the transformer.

Let d_{model} be the embedding dimension of words, and $pos \in [0, L-1]$ be the position of a w word in the $w = (w_0, \ldots, w_{L-1})$ input sequence. Mathematically, the positional encoding of w is defined in Eq. 1.

$$\mathrm{PE}(pos, i) = \begin{cases} \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), & i = 2k \\ \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right), & i = 2k+1 \end{cases} \quad (1)$$

where the positional encoding follows a specific, learned pattern to identify word position or the distance between words in the sequence [39].

MarianCG has no layer normalization embedding. So, positional embedding gets the order of position identifier added to vectors for the transformer to know the order of the sequence.

Marian encoder and decoder architecture

After tokenization and embedding with positional embedding, the next step is to input these embeddings to Marian Encoder and Marian Decoder. Marian Encoder as shown in Fig. 5 is the multi-head self-attention encoder layers connected with layer normalization and after that there are two fully connected layers and final layer normalization.

As the encoder architecture is constructed, we can show and make the decoder in a clear construction. The first steps are tokenization and embedding with positional embedding, and the next step is to input these embeddings to Marian encoder and Marian decoder. Marian decoder as shown in Fig. 6 has the same architecture as the encoder but with adding the encoder attention followed by encoder attention layer normalization. These layers can be added before the two fully connected layers.

The attention component in this Transformer model performs its computations numerous times in parallel. Each of these is referred to as an Attention Head. The Attention module divides its Query, Key, and Value arguments N times and routes each split through a separate Head. The results of all these comparable Attention calculations are then added together to produce a final Attention score. This is known as multi-head attention, and it allows the Transformer to encode many associations and nuances for each word [38].

The following are the main features of the MarianCG construction:

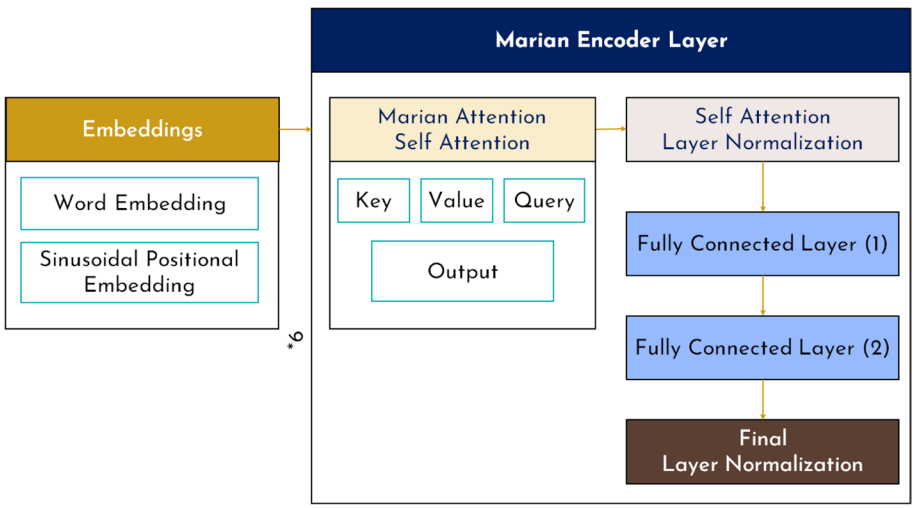


Fig. 5 Marian encoder architecture

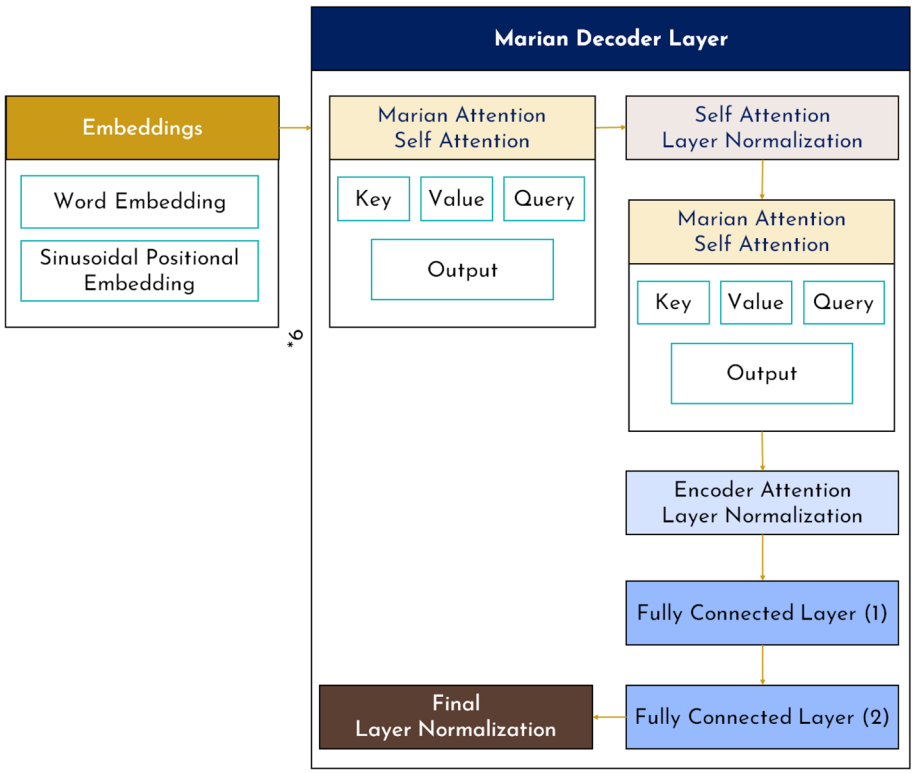


Fig. 6 Marian decoder architecture

1 Marian tokenizer depends on SentencePiece
2 MarianCG contains sinusoidal positional embedding to represent the position of each token
3 No layer normalization embedding for this approach

Datasets

In our research, we used two widely available and well-known data sets, CoNaLa and Django. These datasets were created with the intention of generating code from the corresponding natural language descriptions.

CoNaLa dataset

One of the most common datasets in the code generation task is called CoNaLa [14], and it is created by Carnegie Mellon University NeuLab and STRUDEL Lab. It is called CoNaLa for the name Code/Natural Language Challenge. It has the input as natural language description, with the output as the corresponding python code for this specific input. Table 1 shows some examples of CoNaLa NL-code pairs where the input is the intent which describes the natural language, and the output is the snippet which is the corresponding code for the natural language description.

This dataset has 2,879 annotated NL-code pairings with about 600K mined pairs from over 40,000 distinct stackoverflow questions in the dataset.

Table 1 Intent and snippet examples from CoNaLa dataset

Examples	Intent (natural language)	Snippet (code)
Example 1	Convert a list to a dictionary in python	*b* = dict(zip(a[0::2], a[1::2]))
Example 2	Sort a list of nested lists	l.sort(key=sumnested)
Example 3	How to get the size of a string in python?	print(len('string'))

Table 2 Code generation examples in the DJANGO dataset

Examples	Natural language	Code
Example 1	Define the function do_filter with 2 arguments: parser and token	def do_filter (parser , token) :
Example 2	Convert priority into a floating point integer, substitute it for priority	priority = float (priority)
Example 3	Define the method as_bytes with arguments self and unixfrom set to boolean False	def as_bytes (self , unixfrom = False):

DJANGO dataset

The DJANGO dataset [15] is one of the most commonly used datasets in the code generation task. It has about 19K examples. Each data example is made up of a line of Python code and an annotated natural language description. These examples are divided into 16000 training, 1000 development, and 1805 test annotations in the Django dataset.

Table 2 provides various DJANGO NL-code pair instances. The input is the natural language text, and the output is the corresponding code for this input.

Implementation and experimental work

Evaluation metrics

BLEU score evaluation metric

The BLEU [16] (bilingual evaluation understudy) method evaluates the quality of machine-translated text from one natural language to another. The BLEU statistic counts how many words overlap in a given translation when compared to a reference translation, with successive words scoring better. The connection between a machine's output and that of a person is believed to be quality: "the closer a machine translation is to a professional human translation, the better it is." This is the core notion underlying BLEU.

The BLEU indicator assigns a translation score from 0 to 1; however, it is commonly reported as a percentage number as shown in Table 3. The closer the translation is near 1, the more it resembles a human translation.

Exact match accuracy

This measure is quite simple to compute. This metric is used to compare the similarities and differences between two texts. Equation 2 shows how to calculate and measure this metric for the two sentences $y^{(i)}$ and $\hat{y}^{(i)}$. The first sentence $y^{(i)}$ is the reference sentence, which contains 100% of the genuine needed sentence. The second sentence $\hat{y}^{(i)}$, is the predicted sentence created by the model. Exact match accuracy = 1 if the characters of the model's prediction completely match all the characters of the genuine reference

Table 3 BLEU score

BLEU score	Interpretation
Less than 10	Almost useless
10–19	Hard to get the gist
20–29	The gist is clear, but has significant grammatical errors
30–40	Understandable to good translations
40–50	High quality translations
50–60	Very high quality, adequate, and fluent translations
Greater than 60	Quality often better than human

response; else, exact match accuracy is determined using comparable characters and is in the range between 0 and 1. Exact match accuracy is zero if all characters in the model's prediction do not match all characters in the genuine reference text.

$$ExactMatchAccuracy = \frac{1}{n}\sum_{i=1}^{n}\left[I\left(y^{(i)} == \hat{y}^{(i)}\right)\right] \tag{2}$$

where n is the number of examples, $y^{(i)}$ is the true labels for the i^{th} examples in the reference text, and $\hat{y}^{(i)}$ is the predicted labels for the i^{th} examples.

Implementation

We obtained MarianCG, a novel transformer model based on the pre-trained transformer, by fine-tuning the MarianMT transformer model using the CoNaLa and DJANGO datasets.

For the CoNaLa dataset, we followed [24, 27] and chose the top mined samples based on the likelihood that the NL-Code combination is accurate. The produced CoNaLa dataset had around 13K distinct NL-Code. This is to ensure a fair comparison that if we want to participate in the CoNaLa challenge, so training the model is done by using the conala-train and/or conala-mined datasets, then taking the rewritten intent field from the conala-test dataset as input and generate output from it.

The dataset as mentioned contains about 13K different NL-Code. This dataset contains the conala-train and examples from conala-mined and the 500 examples in conala-test to compare by the same benchmarks as other state-of-the-art contributors. Also, we implement MarianCG to adapt DJANGO dataset which has about 19K pairs of NL-Code.

We noticed high and accurate results on the DJANGO training and testing. So, we decided to do another training process on the CoNaLa dataset with more data and a little batch size. Table 4 displays the datasets employed in each experiment, as well as the dataset size and the number of records in the training, validation, and test sets of data.

Experimental setup

The proposed model was implemented and trained with the dataset using Google Colab Pro service. This allowed us to use 512 input tokens with a batch size of 8. We used Python programming and PyTorch framework to build our model with the HuggingFace transformer module.

Table 4 Datasets in each experiment and distribution of the data

Experiment	Dataset	Dataset size	Dataset split		
			Train	Validation	Test
Experiment 1	CoNaLa	13K	11125	1237	500
Experiment 2	DJANGO	19K	16000	1000	1805
Experiment 3	CoNaLa	26K	24687	1237	500

Table 5 Configuration parameters on the training MarianCG model

Parameter	Value
optimizer	Adam optimizer
Learning rate	$5e^{-5}$
Weight decay	0.01
Maximum position embeddings	512
Number of hidden layers	6
scale embedding	TRUE
Activation function	swish
Learning rate scheduler	Linear
Warmup ratio	0.05
Length penalty	0.9

For training, we depend on the HuggingFace trainer and their implementation of the learning rate scheduler. As well as MarianCG model accepts natural language input and generates Python code in the output. Table 5 lists the configuration parameters that were employed throughout the training phase. For text generation, we adopted the MarianCG model with a linear layer and a distinct bias. Also, beam search and early stopping were employed in the generation phase.

Results

Experiment 1

We began our work by fine-tuning and generating MarianCG transformer model using the CoNaLa dataset. This experiment yields a dataset of around 13K pairings of natural language and code.

Table 6 shows the results of the first experiment, where MarianCG model predictions propelled this model to be one of the top accurate code generation models in terms of accuracy and BLEU score.

Our model got the highest exact match accuracy through all models. This experiment obtained a BLEU score of 30.92, and achieved 6.2 % in the exact match accuracy with the advantage of multi-head attention, and ease to use through the huggingface hub at https://huggingface.co/AhmedSSoliman/MarianCG-CoNaLa.

Experiment 2

In our second attempt, we trained the MarianCG model using another dataset from the code generation challenge, which has more examples. DJANGO is the name of this dataset, which contains 19K natural language and code pairing entries. This dataset is one of

Table 6 Results of the first experiment on CoNaLa

Model	BLEU score	Exact match accuracy	Year
TranX + BERT w/ mined [28]	34.2	5.8	2022
BERT + TAE [26]	33.41	-	2021
External Knowledge With API + Reranking [24]	32.26	-	2020
MarianCG (Ours)	30.92	6.2	2022
External Knowledge With API [24]	30.69	-	2020
BART W/ Mined [27]	30.55	-	2021
Reranker [21]	30.11	2.8	2019
BART Base [27]	26.24	-	2021
TranX [20]	24.3	-	2018

Table 7 Results of MarianCG model on DJANGO dataset

Rank	Model	BLEU score	Exact match accuracy	Year
1	MarianCG (Ours)	90.41	81.83	2022
2	TranX + BERT w/ mined [28]	79.86	81.03	2022
3	BERT + TAE [26]	-	81.77	2021
4	Reranker [21]	-	80.2	2019
5	TranX [20]	-	73.7	2018
6	ipn [40]	77.6	62.3	2016
7	Phrasal Statistical MT [40]	47.6	31.5	2016

the most commonly used for this job, and our implementation and training MarianCG model produced highly accurate predictions.

Table 7 displays the results of the code generation models on the DJANGO dataset. MarianCG is regarded to be the greatest model in the DJANGO challenge. MarianCG has a BLEU score of 90.41 and records an exact match accuracy of 81.83%. A comparison of all values is shown in Fig. 7. Our DJANGO-trained model is available via the huggingface hub, which can be found at https://huggingface.co/AhmedSSoliman/MarianCG-DJANGO.

Experiment 3

We discovered that obtaining additional data resulted in more accurate results. So, for our third experiment, we trained the MarianCG model with an increased amount of training samples. In this experiment, we used the CoNaLa dataset again, but this time with 26K records. This reached our expectations and placed MarianCG model at the top of the CoNaLa challenge. The new testing data results are more dissimilar to our initial trial. This experiment yielded a 34.43 BLEU score and a 10.2% exact match accuracy. Comparing our model to the CoNaLa benchmark models after this experiment revealed that the MarianCG model has the most accurate predictions when compared to other state-of-the-art models. This is displayed in Table 8 which compares the results of this experiment to other models in the CoNaLa code generation challenge, showing the BLEU Score and exact match accuracy of each model.

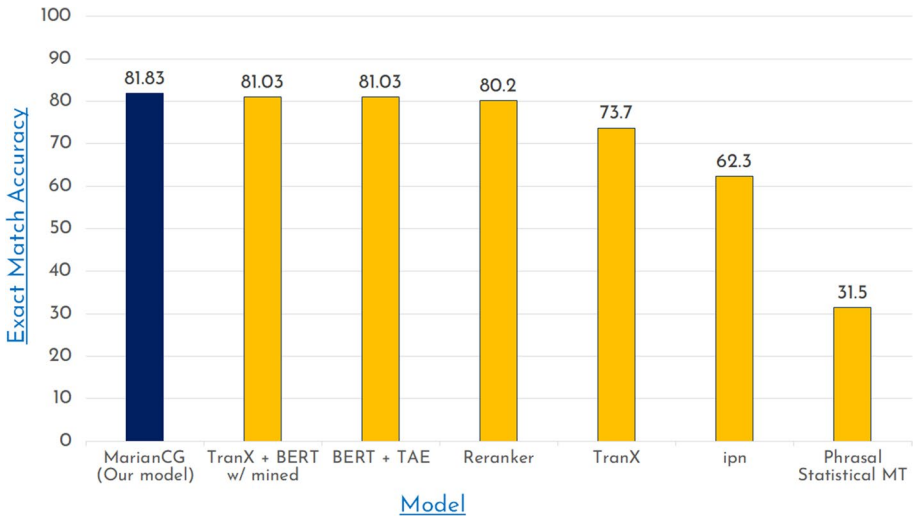


Fig. 7 Results on DJANGO

Table 8 Results of MarianCG model on the CoNaLa dataset

Rank	Model	BLEU score	Exact match accuracy	Year
1	MarianCG (Ours)	34.43	10.2	2022
2	TranX + BERT w/ mined [28]	34.2	5.8	2022
3	BERT + TAE [26]	33.41	-	2021
4	External Knowledge With API + Reranking [24]	32.26	-	2020
5	External Knowledge With API [24]	30.69	-	2020
6	BART W/ Mined [27]	30.55	-	2021
7	Reranker [21]	30.11	2.8	2019
8	BART Base [27]	26.24	-	2021
9	TranX [20]	24.3	-	2018

Also, Fig. 8 depicts all the CoNaLa dataset results. In addition, MarianCG model is available through the huggingface hub at https://huggingface.co/AhmedSSoliman/MarianCG-CoNaLa-Large.

Discussion

MarianCG model is ranked as the first model for its accurate predictions in terms of BLEU score and exact match accuracy. This model has a fewer size architecture. It has six layers in the encoder and six in the decoder, whereas other models have larger model sizes. Table 9 shows the deep learning models employed in the code generation challenge and the number of layers obtained in each encoder and decoder. TranX + BERT trained their model on 100K CoNaLa samples. On the CoNaLa dataset, we trained our model on 13K and 26K records, respectively; hence, the trained data was little compared to others.

This demonstrates that we trained our model with less data than previous SOTA models, and our model is also smaller. As a result of the deep learning architectures and the quantity of the dataset, our model is both fast and accurate.

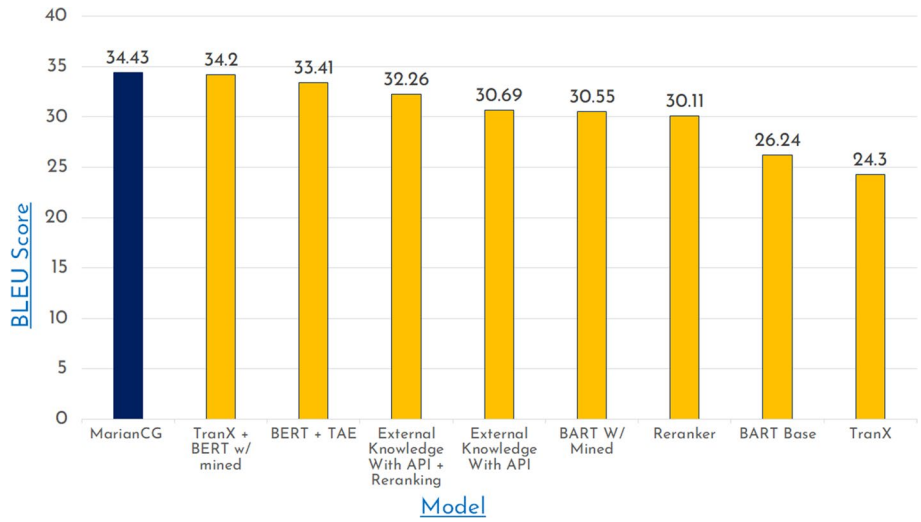


Fig. 8 Results on CoNaLa

Conclusions

For the code generation challenge, we proposed MarianCG, a Transformer language model to predict and generate code from the natural language description. We carried out three experiments in which we developed the MarianCG model on numerous examples in the CoNaLa and DJANGO datasets. This research demonstrated that we are able to employ a machine translation model as a solution model for the code generation problem. Transfer learning enabled us to obtain a precise model for the code generation challenge, where implementation is dependent on fine-tuning a pre-trained language model.

MarianCG was fine-tuned using MarianMT, a machine translation language model that was created with a dependency on the Microsoft Marian toolkit. Our model has the benefit of zero-shot learning, as well as a sinusoidal positional embedding architecture, multi-head attention, and Marian tokenizer depending on SentencePiece. MarianCG received a BLEU score of 30.92 and an exact match accuracy of 6.2% in our first attempt with the CoNaLa dataset. The second experiment was performed on the DJANGO dataset and yielded a BLEU score of 90.41 with an exact match accuracy of 81.83%.

Finally, the third effort used the CoNaLa dataset, but with double the number of examples compared to the first attempt. The final experiment yields excellent predictions, and the MarianCG model rises to the top of the demanding models. MarianCG model achieved a 34.43 BLEU score with a 10.2% exact match accuracy. This model has the advantage of its small size, and it is fast and accurate.

Our long-term goal is to develop a code generation model so that it can predict optimized code with high accuracy and consider code theory topics like SOLID principles and OOP concepts. In addition, we will continue in demonstrating that employing machine translation language models may work well in the code generation task. Furthermore, we will strive toward writing numerous lines of code and demonstrate to show how to generate source code for a certain programming language from another computer programming language. This is known as a code translation task because the

Table 9 Comparing the deep learning models contributed to the code generation task

Model	Encoder		Decoder		CoNaLa dataset			DJANGO dataset		
	Name	Encoder size	Name	Decoder size	Dataset size	BLEU score	Exact match score	Dataset size	BLEU score	Exact match score
MarianCG (Ours)	MarianEncoder	6 layers	MarianDecoder	6 layers	26K	34.43	10.2	19K	90.41	81.83
TranX + BERT w/ mined [28]	bert-base-uncased	12 layers	Grammar based	-	100K	34.2	5.8	19K	79.86	81.03
BERT + TAE [26]	bert-base-uncased	12 layers	Transformer Decoder	4 layers	100K	33.41	-	19K	-	81.77
MarianCG (Ours)	MarianEncoder	6 layers	MarianDecoder	6 layers	13K	30.9	6.2			
BART W/ Mined [27]	facebook/bart-base	6 layers	facebook/bart-base	6 layers	13K + Question bodies	30.55	-			
BART Base [27]	facebook/bart-base	6 layers	facebook/bart-base	6 layers	13K	26.24	-			

input is a programming language such as java and the output is another programming language such as python.

Abbreviations

MarianNMT	Marian neural machine translation
MarianMT	Marian machine translation
MarianCG	Marian code generation
API	Application Programming Interface
BLEU	Bilingual evaluation understudy
OPUS	Open Parallel Corpus
NL	Natural language
CoNaLa	Code/natural language dataset
AST	Abstract syntax tree
BPE	Byte-pair-encoding

Acknowledgements
Many thanks to my family: my wife, father, mother, and brothers for their support. I would like to express my gratitude to my supervisors in this work Prof. Samir Shaheen and Dr. Mayada Hadhoud, and many thanks to Prof. Mohamed Zaki, who guided me a lot throughout my Masters work. I would also like to thank my friends who supported me and offered deep insight into the study.

Authors' contributions
We are aiming to address the code generation problem and create a transformer model that can provide very accurate results. We presented the MarianCG transformer model, which is a code generation model capable of generating code from natural language. This paper discusses the significance of adopting the Marian machine translation model to solve the problem of code generation. In our implementation, we demonstrated that a machine translation model may be used as a code generation model.
We become new contributors and state-of-the-art in tackling this challenge using the CoNaLa and DJANGO datasets based on the model's greatest output predictions, achieving a BLEU score of 34.43 and an exact match accuracy of 10.2% with CoNaLa. In addition, DJANGO has a BLEU score of 90.41 and an exact match accuracy of 81.83%. The structure of MarianCG model includes sinusoidal positional embedding but no layer normalization embedding; the tokenizer depends on SentencePiece.
The authors read and approved the final manuscript.

Funding
No specific funding has to be declared for this work.

Availability of data and materials
All the data used and/or analyzed during the current study are available from the corresponding author upon reasonable request.
● Datasets
We used CoNaLa and DJANGO datasets in our experiments.
1 CoNaLa
The CoNaLa dataset is the code generation corpus from Carnegie Mellon University NeuLab and STRUDEL Lab. It is available at: https://conala-corpus.github.io/.
CoNaLa dataset contains an automatically mined dataset with 600K examples, and each example contains a pair of intent and the corresponding snippet. Additionally, each example obtains more information about the number of questions, how accurate this answer is, and other information. For the test dataset, the creators of this dataset put a benchmark to test on another 500 records for your experiment. We followed other researchers in their work to extract the most accurate examples from the 600K records, with only intent and snippet to work on the code generation problem. This is very helpful to do the number of experiments, and get the results for each experiment. Also, this can be a good thing to let anyone work on this task where he/she doesn't have a powerful GPU.
We have two subsets from the CoNaLa mined dataset:
The first dataset contains 13K records of intent and snippet pairs. It is available at: https://huggingface.co/datasets/AhmedSSoliman/CoNaLa.
2 CoNaLa-Large
This version of CoNaLa has 26K records of intent and snippet pairs. It is available at: https://huggingface.co/datasets/AhmedSSoliman/CoNaLa-Large.
3 DJANGO dataset
It has 19K examples for the code generation task. It is one of the most common datasets in this task, and it is available at: https://github.com/odashi/ase15-django-dataset.
Also, we uploaded DJANGO dataset on the huggingface hub to be available at: https://huggingface.co/datasets/AhmedSSoliman/DJANGO.
● Implementation
Implementation and everything about how implementation was done, datasets, evaluation metrics, notebooks of MarianCG model on the CoNaLa and DJANGO datasets are available in this GitHub repository at: https://github.com/AhmedSSoliman/MarianCG-NL-to-Code.
● MarianCG model
MarianCG models are available now at the huggingface hub, and can be used or tested and integrated with any project. You can find this model and easily deal with various datasets through the following links:

1 MarianCG-CoNaLa https://huggingface.co/AhmedSSoliman/MarianCG-CoNaLa
2 MarianCG-DJANGO https://huggingface.co/AhmedSSoliman/MarianCG-DJANGO
3 MarianCG-CoNaLa-Large https://huggingface.co/AhmedSSoliman/MarianCG-CoNaLa-Large

Declarations

Ethics approval and consent to participate
Not applicable

Consent for publication
Not applicable

Competing interests
The authors declare that they have no competing interests.

Received: 10 January 2022 Accepted: 31 October 2022
Published online: 22 November 2022

References

1. Le TH, Chen H, Babar MA (2020) Deep learning for source code modeling and generation: models, applications, and challenges. ACM Comput Surv (CSUR) 53(3):1–38
2. Han X, Zhang Z, Ding N, Gu Y, Liu X, Huo Y, Qiu J, Yao Y, Zhang A, Zhang L, Han W, Huang M, Jin Q, Lan Y, Liu Y, Liu Z, Lu Z, Qiu X, Song R, Tang J, Wen JR, Yuan J, Zhao WX, Zhu J (2021) Pre-trained models: past, present and future. AI Open 2:225–250. https://doi.org/10.1016/j.aiopen.2021.08.002
3. Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L (2018) Deep contextualized word representations. Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). Association for Computational Linguistics, New Orleans, pp 2227–2237. https://doi.org/10.18653/v1/N18-1202
4. Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805
5. Gu Y, Han X, Liu Z, Huang M (2022) PPT: Pre-trained prompt tuning for few-shot learning. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, vol 1: Long Papers. Association for Computational Linguistics, Dublin, p 8410–8423. https://doi.org/10.18653/v1/2022.acl-long.576
6. Ding N, Qin Y, Yang G, Wei F, Yang Z, Su Y, Hu S, Chen Y, Chan CM, Chen W, Yi J, Zhao W, Wang X, Liu Z, Zheng H, Chen J, Liu Y, Tang J, Li J, Sun M (2022) Delta tuning: a comprehensive study of parameter efficient methods for pre-trained language models. ArXiv arxiv:2203.06904
7. Qin Y, Zhang J, Lin Y, Liu Z, Li P, Sun M, Zhou J (2022) ELLE: Efficient lifelong pre-training for emerging data. In: Findings of the Association for Computational Linguistics: ACL 2022. Association for Computational Linguistics, Dublin, p 2789–2810. https://doi.org/10.18653/v1/2022.findings-acl.220
8. Phuong M, Hutter M (2022) Formal algorithms for transformers. ArXiv arxiv:2207.09238
9. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, et al (2020) Language models are few-shot learners. arXiv preprint arXiv:2005.14165
10. Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I et al (2019) Language models are unsupervised multitask learners. OpenAI blog 1(8):9
11. Shin R, Lin CH, Thomson S, Chen C, Roy S, Platanios EA, Pauls A, Klein D, Eisner J, Van Durme B (2021) Constrained language models yield few-shot semantic parsers. arXiv preprint arXiv:2104.08768
12. Marianmt model. https://www.huggingface.co/docs/transformers/model_doc/marian. Accessed Oct 2021
13. Junczys-Dowmunt M, Grundkiewicz R, Dwojak T, Hoang H, Heafield K, Neckermann T, Seide F, Germann U, Aji AF, Bogoychev N, et al (2018) Marian: fast neural machine translation in c++. arXiv preprint arXiv:1804.00344
14. Yin P, Deng B, Chen E, Vasilescu B, Neubig G (2018) Learning to mine aligned code and natural language pairs from stack overflow. In: 2018 ieee/acm 15th international conference on mining software repositories (msr). IEEE
15. Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, Nakamura S (2015) Learning to generate pseudo-code from source code using statistical machine translation. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp 574–584. https://doi.org/10.1109/ASE.2015.36
16. Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Philadelphia, p 311–318. https://doi.org/10.3115/1073083.1073135
17. Dong L, Lapata M (2016) Language to logical form with neural attention. arXiv preprint arXiv:1601.01280
18. Yin P, Neubig G (2017) A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696
19. Rabinovich M, Stern M, Klein D (2017) Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535
20. Yin P, Neubig G (2018) Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. arXiv preprint arXiv:1810.02720
21. Yin P, Neubig G (2019) Reranking for neural semantic parsing. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Florence, p 4553–4559. https://doi.org/10.18653/v1/P19-1447

22. Shin EC, Allamanis M, Brockschmidt M, Polozov A (2019) Program synthesis and semantic parsing with learned code idioms. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 2019). Advances in Neural InformationProcessing Systems, Vancouver, p 10825–10835. https://dl.acm.org/doi/10.5555/3454287.3455258
23. Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L (2020) Treegen: a tree-based transformer architecture for code generation. Proceedings of the AAAI Conference on Artificial Intelligence, vol 34 No. 05. AAAI-20 Technical Tracks 5, Palo Alto, p 8984-8991. https://doi.org/10.1609/aaai.v34i05.6430
24. Xu FF, Jiang Z, Yin P, Vasilescu B, Neubig G (2020) Incorporating external knowledge through pre-training for natural language to code generation. arXiv preprint arXiv:2004.09015
25. Dahal S, Maharana A, Bansal M (2021) Analysis of tree-structured architectures for code generation. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. Association for Computational Linguistics, Bangkok, p 4382–4391. https://doi.org/10.18653/v1/2021.findings-acl.384
26. Norouzi S, Tang K, Cao Y (2021) Code generation from natural language with less prior knowledge and more monolingual data. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, vol 2: Short Papers. Association for Computational Linguistics, Bangkok, p 776–785. https://doi.org/10.18653/v1/2021.acl-short.98
27. Orlanski G, Gittens A (2021) Reading stackoverflow encourages cheating: adding question text improves extractive code generation. arXiv preprint arXiv:2106.04447
28. Beau N, Crabbé B (2022) The impact of lexical and grammatical processing on generating code from natural language. arXiv preprint arXiv:2202.13972
29. Wang Z, Cuenca G, Zhou S, Xu FF, Neubig G (2022) Mconala: a benchmark for code generation from multiple natural languages. arXiv preprint arXiv:2203.08388
30. Kusupati U, Ailavarapu VRT (2022) Natural language to code using transformers. ArXiv arxiv:2202.00367
31. Al-Hossami E, Shaikh S (2022) A survey on artificial intelligence for source code: a dialogue systems perspective. ArXiv arxiv:2202.04847
32. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078
33. Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L (2019) Bart: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461
34. Subramanyam Kalyan K, Rajasekharan A, Sangeetha S (2021) Ammus: a survey of transformer-based pretrained models in natural language processing. arXiv e-prints arXiv–2108
35. Kudo T, Richardson J (2018) Sentencepiece: a simple and language independent subword tokenizer and detokenizer for neural text processing. arXiv preprint arXiv:1808.06226
36. Kudo T (2018) Subword regularization: improving neural network translation models with multiple subword candidates. arXiv preprint arXiv:1804.10959
37. Sennrich R, Haddow B, Birch A (2015) Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909
38. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems 30 (NIPS 2017), Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA. Curran Associates, Inc., p 5998–6008. https://papers.nips.cc/paper/7181-attention-is-all-you-need
39. Alammar J (2018) The illustrated transformer. http://jalammar.github.io/illustrated-transformer/. Accessed May 2021
40. Ling W, Blunsom P, Grefenstette E, Hermann KM, Kočiský T, Wang F, Senior A (2016) Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Berlin, pp 599–609. https://doi.org/10.18653/v1/P16-1057

Publisher's Note