AUTOMATIC CODE GENERATION FOR COMPUTER VISION APPLICATIONS

By

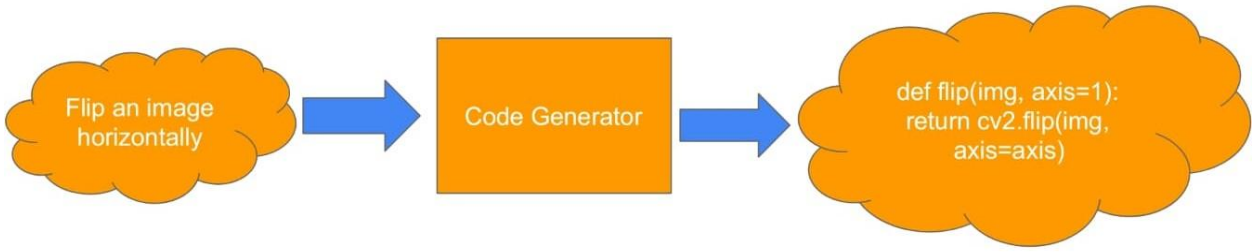
Arpit Kumar Rai
BTech, CSE (200190)

Mehta Shrey Kartik
BTech, CSE (200580)

Supervisor:- Dr Priyanka Bagade
Course:- CS396A

1. Introduction

Code generation can be viewed as a machine translation problem where a natural language input provided by the user is converted into a programming language output.



The above figure illustrates an example of code generation where a user has an image and wants to flip it horizontally. All she has to do is give a natural language command, “Flip an image horizontally”, and the code generator will automatically write the code to perform this task.

Recently, many text-to-code generation models have been developed that parse the given input to generate code as the output. The three most popular models in this domain are **Codex,** developed by OpenAI; CodeT5, acquired by Salesforce; and **AlphaCode,** developed by Deepmind. Although all these three models work towards a similar goal, they vary in the methodologies they have developed and the accuracy of their performance in various tasks.

After a deep analysis, we found some limitations of the model while generating code related to computer vision and deep learning tasks. Our primary aim was to produce a model which generates (mostly) valid code for computer vision tasks by fine-tuning the pre-existing open-source model CodeT5. However, we faced certain limitations to this approach and opted for a more general custom-made transformer approach that we define further in this report.

We present here an example of a use case where CodeX could not provide functionally correct code.

Plotting Errors:

```
plt.imshow(img)
plt.show()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-878eb3b29097> in <module>()
----> 1 plt.imshow(img)
      2 plt.show()

                                  5 frames
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py in set_data(self, A)
    692                 not np.can_cast(self._A.dtype, float, "same_kind")):
    693             raise TypeError("Image data of dtype {} cannot be converted to "
--> 694                             "float".format(self._A.dtype))
    695
    696         if not (self._A.ndim == 2

TypeError: Image data of dtype object cannot be converted to float
```

2. Transfer Learning and Choosing the suitable model

We started to approach this problem using the idea of transfer learning, as we will have a pre-trained model, having the weight and the bias values pre-initialised based on the dataset it is trained upon. This will help us cater to our purpose even if the dataset is small.

So, we searched for the different available models used in the industry, like CodeT5, CodeBART, PyMT5, etc. and found out that the CodeT5 model, which is a code generation model by OpenAI, is the best match due to the following reasons:

- It has code-specific training objectives, trained on the CodeSearchNet dataset, comprising 2 million data points with NL to six different programming languages.
- It has a publicly available fine-tuning code, which we can use to achieve our computer-vision-specific code generation task.
- It uses a transformed-based encoder-decoder architecture, which is very useful when dealing with variable-length input strings of the natural language.

3. CodeT5

3.1 What is CodeT5?

CodeT5 is a model based on a T5 encoder-decoder that can assist in converting the NL code to the appropriate PL code. It incorporates the information specific to the code from the code using an identifier-aware pre-training target.

Pretraining: The CodeT5 model has been pre-trained on the CodeSearchNet dataset, which has about 2 million training data points made up of the output of programming languages (pre-trained on six different programming languages, including Python) and the natural language prompt that corresponds to it.

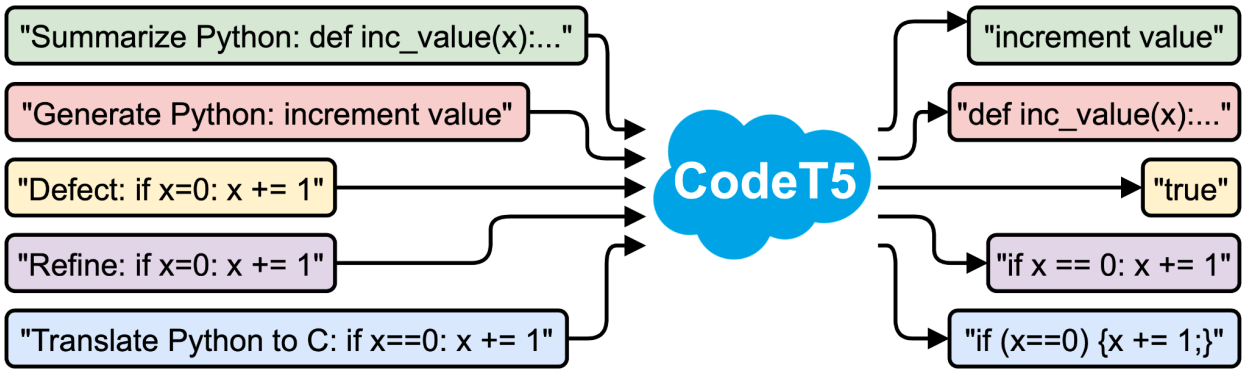
3.2 Advantages of Codex

The CodeT5 paradigm has the following benefits over the Codex:

- The open-source model known as Codex cannot be used for research. Although it can be duplicated using the Codex research paper, this is not practical for specific research needs because it requires many GPU clusters. In turn, CodeT5 is an open-source model that transfer learning can utilise further for tuning.
- Additionally, while collecting data points for the code generation task is challenging, CodeT5 helps compensate for the smaller number of data points in our dataset because it has already been trained on a very sizable dataset.
- Additionally, the model can be improved with less training time because there is less training data.

3.3 Why choose CodeT5?

Among all the code-generating available models, CodeT5 should be chosen for the reasons listed below in addition to the benefits over Codex that were previously mentioned: As we can use the current model and modify it for particular tasks using our training data, it offers a fine collection of resources for further research. It is observed to have outperformed the other encoder- and decoder-only models on criteria like the BLEU score. Due to its pre-training with identifier awareness, it has a superior comprehension of code semantics.



4. Dataset

We could not locate substantial datasets for training on any internet platform due to the highly specialised nature of our challenge, which involved text-to-code generation (in Python) for computer vision problems. Because of this, we manually produced the dataset by pulling (doc_string, generated_Python_code) pairs from GitHub and, to a significant extent, by querying pertinent doc_strings to Codex to obtain the generated Python code. We currently have a dataset with more than 1700 data points, and we intend to expand it further for better training.

5. Results

5.1 Evaluation Metrics

The following metrics were taken into consideration while evaluating the model :

BLEU (Bilingual Evaluation Understudy): BLEU is a metric commonly used to evaluate the quality of machine translation systems. It compares the generated output with one or more reference translations and computes a score based on the n-gram overlap between the work and the reference translations. BLEU score ranges from 0 to 1, with a higher score indicating a better translation quality.

CODEBLEU: CODEBLEU is a variant of BLEU designed explicitly for evaluating the quality of code generation systems. Like BLEU, it measures the n-gram overlap between the generated code and one or more reference codes. Still, it also considers the structural similarity between the generated code and the reference code. CODEBLEU is particularly useful in evaluating the quality of code generation systems that produce code in multiple programming languages.

EM (Exact Match): EM is a metric measuring the percentage of generated outputs that match the reference output. It is commonly used in natural languages processing tasks, such as question-answering and text summarization. Unlike BLEU and CODEBLEU, which consider partial matches between the generated and reference outputs, EM requires an exact match. EM score ranges from 0 to 1, with a higher score indicating a better match between the generated and reference outputs.

5.2 Experiments Carried Out

1. CodeBLUE Score:

We have done experiments on the NL-PL dataset by splitting it into train, validation and test data in an 80:10:10 ratio by randomly choosing among all the data points and breaking them in the ratio given.

The results obtained are as follows:

No. of Epochs	CodeBLEU Score
4	0.49
8	1.29
12	3.04
16	27.6
20	32.15

Table 1

Table 1 shows the BLUE Score corresponding to the number of epochs on the test data in case of splitting the dataset randomly. In this case, the BLEU score obtained on the test set comes out to be 34.30, the EM score comes out to be 25.00, and the CodeBLEU score comes out to be 48.00.

We chose to split the dataset randomly because when we broke the dataset without randomising, we had specific NL data points which were only present in the test data and not in the train data and vice versa.

For, eg. The code corresponding to the NL: “ Crop the image to half its a size ” is not present in the training dataset, but when the asked the corresponding PL output to it by the test data, it gives erroneous results.

So, it is clear that we must randomly pick the data points from the dataset so that we don't face such issues.

6. Salient Features of the Model

Currently, our model produces the PL output corresponding to the NL output with a CodeBLUE score of 1.88 on the dataset of around 1700 data points.

We are using the CodeT5_base model for making the predictions using AWS Services.

The total running time of the model is five h 55min, for the training, development/validation phase and testing (producing the output).

A snippet of the output produced by the model is shown below:

NL input	PL actual	PL predicted

Randomly flip an image horizontally	`def flip_image(image, label):`
	`  if random.random() < 0.5:`
	`  image = cv2.flip(image, 1)`
	`  label = cv2.flip(label, 1)`
	`  return image, label`
		`def flip_image_horizontally(img, axis=1):`
		`  return cv2.flip(img, axis=axis)`

Crop Image randomly	`def crop_image(img,crop_size):`
	`  h, w, _ = img.shape`
	`  top = np.random.randint(0, h - crop_size[0])`
	`  left = np.random.randint(0, w - crop_size[1])`
	`  bottom = top + crop_size[0]`
	`  right = left + crop_size[1]`
	`  return img[top:bottom, left:right, :]`
		`def crop_image(image, x_min, y_min, x_max, y_max):`
		
		`  """ Crop the image with the min and max points`
		`  :param image: image to crop`
		`  :param x_min: min x`
		`  :param y_min: min y`
		`  :param x_max: max x`
		`  :param y_max: max y`
		`  :return: cropped image  """`
		`  return image[y_min:y_max, x_min:x_max]`

We did niot get expected results on training the model, wo we started to look out for other approaches to solve this problem.

7. GNNs for Code Generation:

Since, we did not get expected results from the transfer learning mechanism, we searched for different approaches to this problem statement. One of the approaches we could look out for was using Graphical representation for the structure of the code, so as to remove any errors in the code structure obtained from the ML model.

So, we surveyed through several research papers and surveys based on how to convert text-to-code using GNNs, and found papers related to code summarization and code refactoring using GNNs. The major motivation of using this approach was due to the fact that:

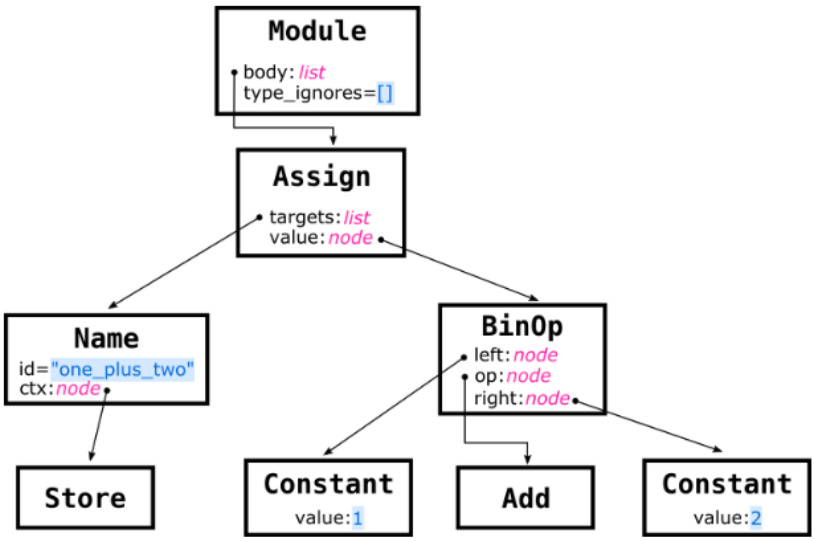
- GNNs easily help us in capturing the complex relationship among different code elements and the input natural language sequence.
- GNNs are time efficient as we can traverse to any node in O(log(n)) time and are also easy to train for larger codebases.

The basic idea behind using the GNNs for code generation is that we can model the code elements as the nodes of the AST, and the edges between these nodes would represent the relationship between these nodes. This relationship can be syntactic as well as semantic.

So, we tried to visualize a python code, which is our target language, into an AST, so that we can use it for the purpose of training the model. We used the *ast* module of python to visualize a snippet of code, and we found the following result:

```
>>> import ast
>>> code = "one_plus_two = 1+2"
>>> tree = ast.parse(code)
>>> ast.dump(tree, indent=4)
```

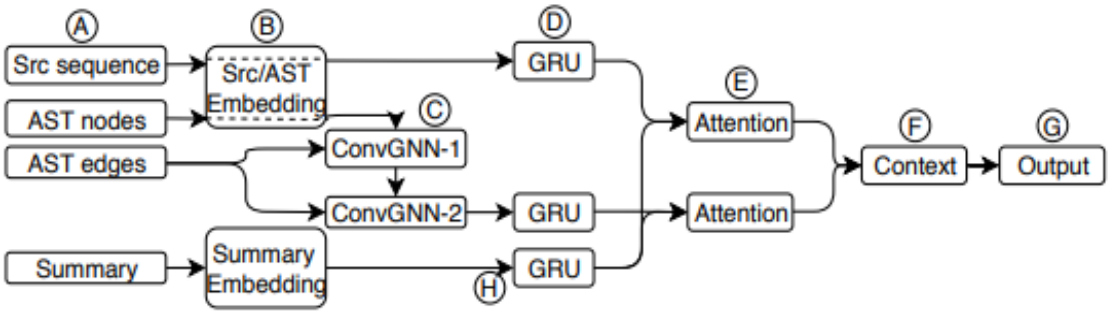
(Code snippet to get the AST)



(Generated AST corresponding to the code snippet)

As can be seen from the AST obtained above, we can see how syntactic relationship is established among the code elements on the basis of the grammar of the Python language, and hence, we found this a useful tool for generating the ASTs corresponding to the PL datapoints in our dataset.

In our survey through the research papers, we found methods that have been used for the task for code summarization, so we thought that we could follow the same architecture followed by the research paper and reverse the task, i.e. feed a natural language sequence to the model and try to obtain the AST from the model, and then convert it to the corresponding python code. The architecture of the neural networks used in that paper was:



(Architecture of the Code Summarization model using GNNs)

We went forward with the approach, but later realized the difficulties in the task, like:

- Ambiguity in natural language: The source code are usually well-structured and are syntacting very strong, while the natural language can be ambiguous with multiple interpretations of the same text. So, the model we design will have to parse the natural language input sequence, understand it's interpretation and generate a source code corresponding to it. This is a challenging task also because we require the source code generated to be syntactically and semantically correct, which requires a good amount of knowledge of the programming language by the ML model.
- Generating AST from source code: The AST is crucial for activities involving code generation because it offers a structured representation of the code that GNNs can quickly understand. However, because it necessitates a thorough comprehension of the programming language's syntax and semantics, building the AST can be challenging. Making an AST for GNNs can be done in a number of ways. To directly extract the syntax tree from the source code, one method is to employ parsing techniques. To build the AST, which can capture the program's higher-level structure, another method is to combine parsing and semantic analysis.

So, we realized that we should stick to the idea of using the transformer architecture itself, for achieving this task of code generation, with varying the different layers, attention mechanisms and optimizers.

8. Various Attention Mechanisms

We realized that the problem statement that we are dealing with is very specific in nature that is limited only to code generation for computer vision applications. Hence we decided to change the lower level architecture of the transformer model, such as the number of encoder/decoder layers, the type of attention mechanism used and the optimizers which could prove beneficial for our use case.

However the code base of CodeT5 is not publicly available, hence preventing us from applying any such changes. Additionally, CodeT5 uses vanilla self-attention mechanism which suffers in identifying long range relations in the input/output sequence. We found that in our usage scenario, the output program that is generated may be significantly longer than the input natural language prompt, which might be difficult to handle effectively using vanilla self-attention mechanism.

All these factors motivated us to write a custom-made transformer model best suited for our purpose. We tried the following attention approaches:

1. **Vanilla Self-Attention**
   This is the simplest form of self attention in which each token in the input sequence attends to every other token in the sequence to generate the output embedding. There are two major disadvantages associated with vanilla self-attention:

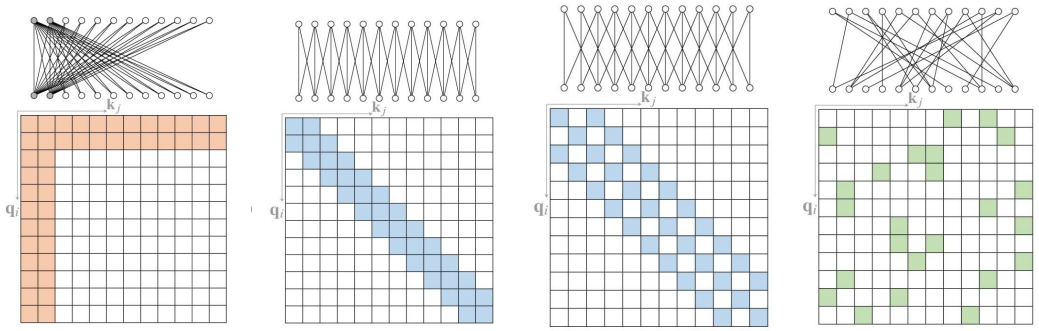
a. **Attention Computation:** Computing attention is an expensive task both in terms of time and memory. Attention on a sequence of length L requires O(L^2) both in time and space complexity. This reduces the capacity of vanilla self-attention to handle long sequence lengths and capture relations between tokens that are far apart.
b. **Large number of Layers:** While computing vanilla self-attention, we need to store the activation value of each layer as these values are required during the back-propagation step. Hence a model with N layers consumes N-times larger memory than a model with a single layer.

1.

2. **Sparse Attention**

This mechanism uses structural bias to limit the number of query/key pairs that each query attends to, as it is observed that for the trained Transformers the learned attention matrix A is often very sparse across most data points. This significantly reduces the runtime complexity of the attention step and allows the model to handle long sequences more effectively. We considered two different classes of sparse attention mechanisms namely:

a. **Structured Attention:** We observed some structure in the sequence and used predefined patterns such as Global Attention, Band Attention and Dilated Attention to identify the relevant tokens that need to be attended to correspond to each word.
b. **Random Attention:** In this scheme, we randomly pick a subset of tokens from the entire set of tokens in the sequence and use them for the attention step.



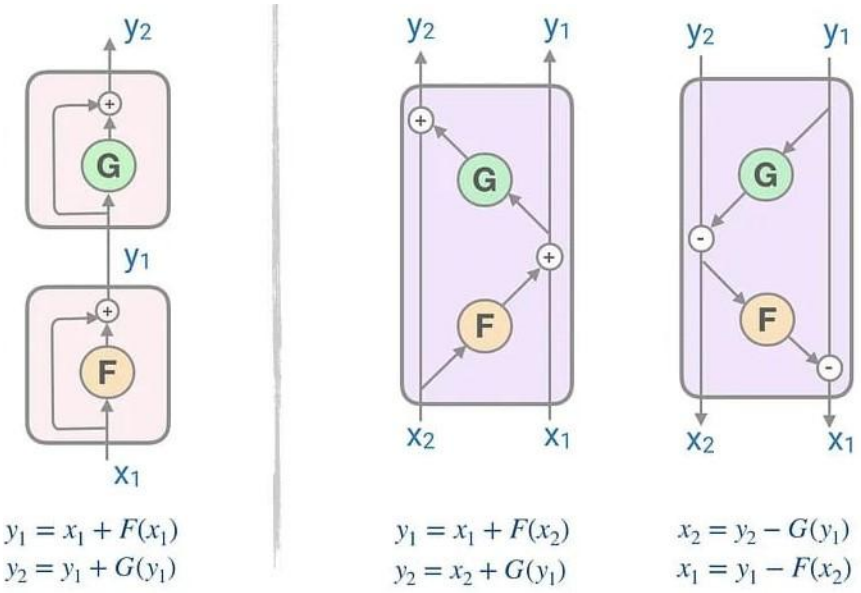
3. **Reformer**

This model tries to overcome both the issues that are present in the Vanilla Self-Attention mechanism.

a. **Reducing Time Complexity:** We used Locality Sensitive Hashing to partition the set of tokens in a sequence into different subsets (buckets). Locality Sensitive Hashing is a hashing algorithm with the property that two passes (represented as their N-dimensional embedding vectors) belong to the same bucket, i.e. have the same hash value if and only if they are closely spaced in the N-dimensional

vector space. Then for each token, we attended only to those other tokens which belonged to the same bucket. This significantly reduced the number of tokens, allowing for larger frame size.

b. **Reducing Memory Footprint:** We used Linearized Attention Model to reduce memory consumption during backpropagation. With Linearized Attention, we use a reversible attention block; hence we do not need to store the activations for each attention layer, as the previous activations can be calculated using the current activation values while performing the backpropagation step itself.



Here F and G represent the Attention and Fe Feed Forward layers respectively. The left part shows the traditional Attention block while the right part shows the Linearized Attention Block. During backpropagation, given the current activation values y1 and y2, the previous activation values x1 and x2 can be recovered quickly in Linearized Attention but not in Traditional Attention.

9. Challenges and Future Work

1. We observed that the results we achieved using Sparse Attention and Reformer models were less promising than expected.
2. We attribute this to the small dataset size available to us during training. We used the same dataset of 1700 data points to train our custom-made transformer model that we used to fine-tune the CodeT5 model.
3. Thus the main focus of our future work is to expand our dataset size.
4. Additionally, we also plan to try out new models and multiple variations of our custom-made transformer model to choose the one that is best suited for our task.
5. Going into the project's next phase, we also need to do the classification task for which a dataset must be made.

10. References

1. https://towardsdatascience.com/beyond-codex-a-code-generation-model-that-you-can-train-6ac9bdcba07f
2. https://openai.com/blog/openai-codex/
3. https://github.com/salesforce/CodeT5
4. https://blog.salesforceairesearch.com/codet5/
5. https://towardsdatascience.com/illustrating-the-reformer-393575ac6ba0
6. https://ai.googleblog.com/2021/03/constructing-transformers-for-longer.html?ref=assemblyai.com