

Deep Learning for Natural Language and Programming Language Processing

Code Comment Generation

A Dissertation Presented for the
Course work EE997: Individual MSc Project for
MSc in Machine Learning and Deep Learning
The Department of Electronic and Electrical Engineering
The University of Strathclyde, Glasgow

Submitted on August
19, 2022

Author :


Syed Junaid Iqbal (202168462)

Supervisor :

Prof. Feng Dong

Declaration of Authorship

I, Syed Junaid Iqbal, hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of Professor Feng Dong

Signed: 

Date: 19 August 2022

Abstract

Syed Junaid Iqbal

Deep Learning for Natural Language Processing and Program Language Processing

Deep learning is a subset of Artificial Intelligence capable of learning semantic features of a specific pattern. In this project, I have tried to explore and exploit the various application of machine translation. Machine translation was originally designed to translate from one natural language to another. Now with the advancement of transformers and better computing, there are ways to translate between natural language and program language. Some common tasks of natural language and program language include code-to-text, code-to-code and text-to-code.

My research revolves around code-to-text translation, where given any code snippet the model is trained in such a manner that it can generate comment. In order to achieve this I have explored both the legacy method of using RNN and LSTM based encoder-decoder model and also some of the state of the art pretrained model like Code BERT and Code T5 models.

For this project the proposed way of using Code BERT has been implemented. The formal evaluation and observation of the comment generated by model, we can say that Transformers can understand the semantics of a code structure to create meaningful comments.

Acknowledgements

I want to thank Prof. Feng Dong, my supervisor, for his advice and assistance throughout this challenging project. I am grateful for the suggestions made by Prof. Feng Dong during the weekly presentation, which helped me structure my report.

Table of Contents

1	Introduction	1
1.1	Overview	1
1.2	Outcomes	1
1.3	Report Structure	1
2	Background and Related Work	2
2.1	Deep Learning	2
2.1.1	Artificial Neural Networks	2
2.1.2	Training a Model.....	3
2.1.3	Transfer Learning.....	5
2.2	Natural Language Processing	6
2.2.1	Evolution Of Language Modelling	6
2.2.2	Language Model Development Life Cycle	15
2.2.3	Application Of Language Modelling	18
3	Problem Specification	19
3.1	Code Comment Generation	19
3.2	Use of Pretrained Models	19
4	Model Design	21
4.1	Forward Propagation.....	22
4.2	Search Mechanism	23
4.3	Loss Function and Optimiser	23
4.4	Evaluation of Results	24
4.4.1	BLEU score	24
4.4.2	Similarity Score	24
5	Implementation Details	26
5.1	Dataset	26
5.2	Data Pre-Processing	27
5.3	Create Input IDs and Attention Mask	28
5.4	Pytorch Dataset and Data Loader.....	29
5.5	Load Pre-Trained Model	29
5.6	Setting Up Call-Backs.....	29
5.7	Model Training and Evaluation	30
6	Results and Evaluation	32

6.1	Model Evaluation	32
6.2	Generated Comments	34
7	<i>Implications and Future Work</i>	39
7.1	Summary	39
7.2	Limitations and Future Work	39
8	<i>Bibliography</i>	40
9	<i>Further Implementation Details</i>	1
9.1	Hardware	1
9.2	Languages and Frameworks	1
9.3	Data Pre-Processing	1
9.3.1	Removing Docstring from The Code	1
9.3.2	Handling Outliers	2
9.4	Data Loader	2
9.5	Loading Pre-Trained Models	3
9.6	Function to generate comments for test cases	4

Chapter 1

1 Introduction

1.1 Overview

This research work explores the capability of an AI system to explore and understand the semantics of programming language. Here I will develop a model that uses a pre-trained state-of-the-art model to generate the comments for a programming language. I will explore various deep learning approaches to solve code-to-text translation in this project. The use of Transformers for natural language tasks such as machine translation, image captioning, and sentiment analysis has yielded promising results. Speaking about program language, many factors are to be considered, such as return type, indentations, nested functions, and others, to name a few. We will also use newly developed frameworks, such as Pytorch lightning to automate the processes involved in developing a deep learning model.

1.2 Outcomes

The proposed Code BERT model has been implemented in this project to solve code-to-text translation. Here we generate comments for Python and JavaScript languages. We can see that the model can comprehend the semantic structures of code through a formal evaluation and by observing the comment that was produced. The model can generate insightful comments for the given code snippets.

1.3 Report Structure

First, the report presents underlying concepts of deep learning that will be required to understand the use of transformers better. A detailed study of natural language processing and program language processing such that we can correlate the problem statement better. The model design, and implementation detail, followed by an analysis of results, are discussed toward the end of this report. The appendices will mention all the code snippets and the software used.

Chapter 2

2 Background and Related Work

This chapter revolves around the fundamental idea and concepts of Deep Learning. Here I will cover the history, evolution, and current state-of-the-art techniques and architecture related to this project. Moving on in this chapter, we will have an in-depth understanding of the encoder-decoder-based models using Transformers.

2.1 Deep Learning

Deep learning is a more prominent family of machine learning techniques built on artificial neural networks and representation learning. Deep Learning is capable of unsupervised, semi-supervised, and supervised learning. In disciplines like computer vision, speech recognition, natural language processing, and machine translation, the use of deep learning architectures like deep neural networks, deep reinforcement learning, recurrent neural networks, convolutional neural networks, and Transformers have generated results comparable to and in some cases, surpassing those of human expert.

This section will cover the basics of deep learning, including understanding neural networks, model training, optimisation, loss function and other training processes.

2.1.1 Artificial Neural Networks

Artificial neural networks(Wang, 2003) are biologically inspired models. Perceptron is the most basic form of an artificial neural network. Perceptron can be defined as a product of a weighted sum of inputs and a nonlinear activation function. If $X_1, X_2, X_3, \dots, X_n$ Are the inputs and $W_1, W_2, W_3, \dots, W_n$ is, the corresponding weights then weighted sum is given as $X_1W_1 + X_2W_2 + X_3W_3 + \dots X_nW_n$, finally, we pass the weighted sum through a nonlinear activation function σ . The perceptron is represented by equation 2.1

$$\hat{y} = \sigma (\sum_{i=1}^n x_i w_i)$$

Adding multiple perceptrons(Ramchoun et al., 2016) in terms of depth and width, we create a dense feedforward layer used as an ANN. Unlike perceptron, ANN can have multiple outputs, as shown in Figure 2.1. With every hidden layer, we can extract more features. An ANN with numerous layers is referred to as a deep neural network. Multiple layers are used in the model to help it better understand the issue it is attempting to address.

If we use a three-layer network as a simple example, the input layer is the first layer, the output layer is the last, and the middle layer is referred to as the hidden layer. We use the input layer to receive our data input and the output layer to get our output class probability. To make the model more complex and better suit our needs, we can

increase the number of hidden layers as much as we like. Hidden layers can be grown in terms of both depth and width. ANN is limited to the classification task. Depending on the type of activation function that will be used, the values are clipped to match the range.

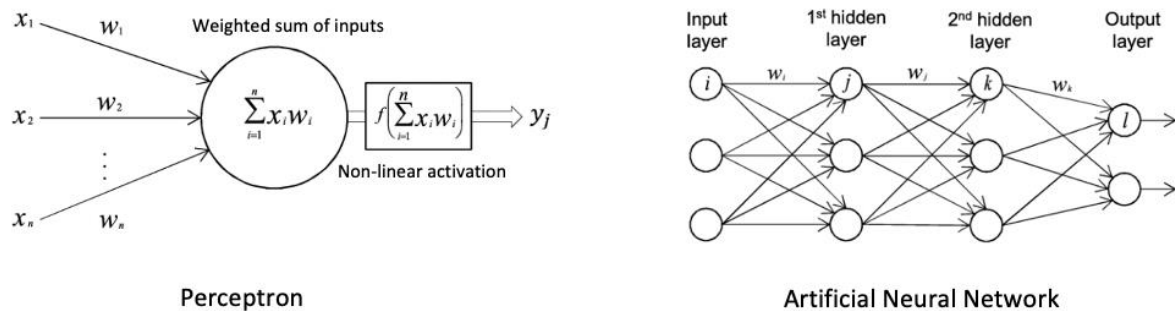


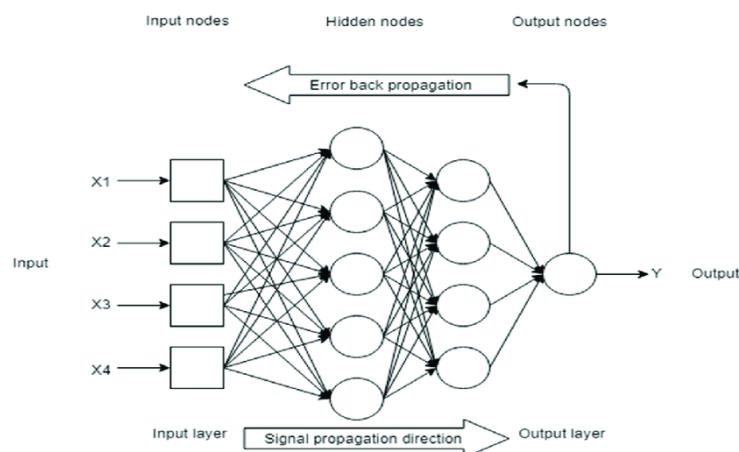
Figure 2. 1: Schematic structure of perceptron (Left) and artificial neural network (Right) (Vieira et al., 2017)

2.1.2 Training a Model

Deep learning models without adequate training (Ramchoun et al., 2016) give random values. Models are trained by adjusting the weights. A model is often trained through supervised learning, in which the model's weights (or parameters) are changed to enable it to produce the desired result given an input. When we train our model using supervised learning methods, we calculate the loss and adjust the weights accordingly. There are three basic steps involved in model training "Forward propagation", "Calculate loss", and finally "Backward propagation".

2.1.2.1 Forward Propagation

As the name suggests, the data is fed to the feedforward layer (Svozil et al., 1997) through the input layer. It is essential to make sure that the information does not flow back in the backwards direction. During the forward propagation, input data is fed through ANN to generate the desired output, as shown in Figure 2.2. During this process, two processes happen simultaneously. First is calculating the weighted sum of inputs at every neuron, which is passed through a nonlinear activation function (Murat H., 2006; Svozil et al., 1997). This is done to add non-linearity to our neural network. Some of the most common activation functions used are SoftMax, Sigmoid, tanh, and ReLu.



2.1.2.2 Calculate Loss

Loss can be defined as the difference between actual and predicted values. Loss helps us evaluate the model's performance. If the model has a significant loss, it means the model predicts the values that are not close to the ground truth. While training our model, we seek to minimise the loss (or error). To understand better, let's consider an example of binary classification. If $y_1, y_2, y_3, \dots, y_n$ are the actual values and $\hat{y}_1, \hat{y}_2, \hat{y}_3, \dots, \hat{y}_n$ are the predicted values, then the error is given as

$$error = y_1 - \hat{y}_1 + y_2 - \hat{y}_2 + y_3 - \hat{y}_3 + \dots y_n - \hat{y}_n$$

$$error = j(\theta) = \frac{1}{n} \sum_{i=1}^n (y_n - \hat{y}_n)^2$$

The above equation is called mean square error (Karunasingha, 2022). It is the sum of the square difference between the actual and predicted values. Once we have calculated the error, we must choose the appropriate learning rate for our optimisation algorithm. As we have many trainable parameters, it is impossible to instantaneously get the optimal weights for our model. To get the optimal weights models, depend on a search or optimisation algorithm to reduce the loss or error. We have various optimisation algorithms based on the data's type and scale. Gradient descent, Stochastic gradient descent, and Adam are some popular optimisation techniques. The optimisation algorithm is responsible for random weight initialisation. Once the weights have been initialised, we constantly update the old weights using gradients, learning rate and old weights. Let us consider w_0 as randomly initialised weight, α as our learning rate, and $j(\theta)$ our loss function. We can represent the gradient descent algorithm as shown in the equation

$$w_0 = w_0 - \alpha \frac{\partial j(\theta)}{\partial w_0}$$

The optimisation (Graña Drummond & Svaiter, 2005) algorithm aims to find the global minima. This is where the learning rate plays a curtail role. If the learning rate is set to a very high value, we might overshoot global minima and move in the wrong direction. According to this, if the learning rate is minimal, it will take considerably longer to reach the minimum. This is usually a hyperparameter. The best way to achieve the optimal learning rate is by using a learning rate scheduler (Senior et al., 2013), where the model automatically updates the learning rate based on the descent rate. The model reaches the global minima most efficiently when we have an optimal learning rate. We can see this in the figure below.

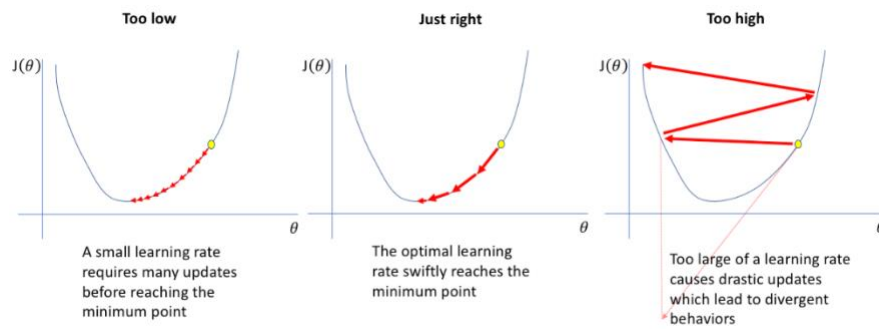


Figure 2. 3: Effects of Learning Rate on Optimisation Algorithm (JEREMY JORDAN.)

2.1.2.3 Back Propagation

The foundation of neural - network training is back-propagation(Ramchoun et al., 2016; Svozil et al., 1997). As we see in figure 2.3, adjusting a neural network's weights is based on the error rate (or loss function) relative to the previous epoch (i.e. iteration). Lower error rates are ensured through proper weight adjustment, which broadens the model's applicability and makes it more dependable. The critical component of Back Propagation is calculating the gradients of the loss function concerning the single weight and then applying the chain rule. The weight of every neuron is updated using a chain rule.

We can categorise backpropagation (HECHT-NIELSEN, 1992) into two types they are “static back-propagation” and “recurrent back-propagation”. A static input in this network maps to a static output. Static backpropagation will work well for static classification issues like optical character recognition. Recursive backpropagation is used up until a predetermined threshold is reached. The error is estimated and transmitted backwards after the threshold. Static backpropagation seems to be as quick as static mapping, distinguishing these two methods.

2.1.3 Transfer Learning

A model created for one task is used as the basis for another using the machine learning technique known as transfer learning (Wolf et al., 2019) . Pre-trained models are frequently utilised as the foundation for deep learning studies in computer vision and natural language processing. They save time and money compared to developing models from scratch and tend to perform better on related tasks. We can see the transfer of weights from one model to the other in figure 2.4.

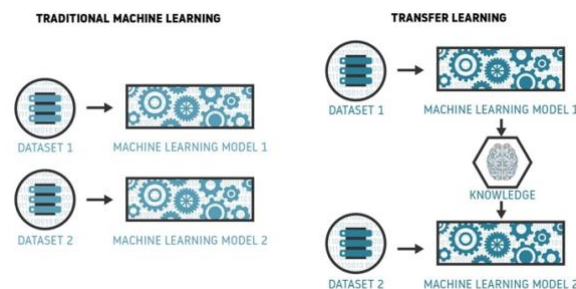


Figure 2. 4: Illustration of Transfer Learning (Diego Lopez Yse, n.d.)

We can achieve transfer learning in one of two ways. First, we train our model on extensive data with high variance data, and then we save the model and try reusing it on a similar type of problem. The other way is by fine-tuning (Vrbancic & Podgorelec, 2020) a pre-trained model. It is essential to keep in mind that the model performs the best when using the similar type of data that the model was used for training. During the transfer learning process, the model weights remain unchanged. Only the importance of the final predictive layer is updated accordingly.

2.2 Natural Language Processing

Natural Language Processing (Nadkarni et al., 2011) is a subfield of machine learning that deals with analysing, comprehending, and creating the languages humans naturally use to communicate with other humans. Solving the NLP task is challenging as it involves various parameters to be considered, like understanding the context of the word in a sentence, the correlation between the past and the future expression in a sentence and overcoming the noise factor in the text; this can be ambiguity, colloquial and slang, synonyms to name a few.

In this section, I will be talking about the evolution of language modelling, the various steps involved in language modelling and the different application in the industry.

2.2.1 Evolution Of Language Modelling

Language modelling has been around for more than half a century now. Over the decades, we see there has been a dramatic shift in the techniques that have been used to process language models. There are two types of language models, “statistical language models” and “Neural Language Models”. Probabilistic models can predict the following word in a series in statistical models based on the previous comments. Some techniques are N-grams and Hidden Markov Models (HMM). On the other hand, natural language models use neural networks to extract features. In recent times we can see that Natural Language based models have surpassed humans in a specific domain.

2.2.1.1 Bag-of-Words Model

The statistical language modelling technique is used to analyse the textual data in a document based on the reoccurrence of words. The method is straightforward and adaptable and may be applied in various ways to extract document features. A bag of words (Qader et al., 2019; Zhao & Mao, 2018) is a textual illustration that shows where words appear in a manuscript. It entails two components: first, a vocabulary of well-known phrases, and second, a measurement of the vocabulary's prevalence. It is referred to as a "bag" of words because any details regarding the arrangement or structure of the words within the document are ignored. The model doesn't care where in the document specific terms appear; it is only interested in whether they do.

	about	bird	heard	is	the	word	you
About the bird, the bird, bird bird bird	1	5	0	0	2	0	0
You heard about the bird	1	1	1	0	1	0	1
The bird is the word	0	1	0	1	2	1	0

Figure 2. 5: Illustration of the bag-of-words model (Alexis Perrier, n.d.)

There are three steps involved in the Bag-of-Words model. First, we tokenise the sentences and build our vocabulary. Vocabulary is a total collection of words that were present in the corpus. In the next stage, we count the total number of times that term has been repeated in each sentence. This will create a matrix-like structure, as shown in figure 2.5. To use this as a classification model like a spam filter, we will count the number of times the words repeat. Here if the number of words that resemble spam words is greater than the number of comments from non-spam, then we conclude the message is spam. As we can see here, no context or semantics of the terms are considered. The bag of word model is simple and efficient if the data or vocabulary is small. As the size increases, this approach is highly inefficient. Another drawback is that if the same word repeats itself, the multiple times model tends to be more biased.

2.2.1.2 Term Frequency-Inverse Document Frequency

A statistical technique called TF-IDF (term frequency-inverse document frequency) (Amir Sjarif et al., 2019; Christian et al., 2016; Havrlant & Kreinovich, 2017) assesses how significant a word is to a document within a collection of documents. A word's frequency in a document and its inverse document frequency over a group of documents are multiplied to achieve this.

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

	Doc 1	Doc 2	...	Doc n
Term(s) 1	12	2	...	1
Term(s) 2	0	1	...	0
...
Term(s) n	0	6	...	3

Figure 2. 6: Calculating words TF-IDF score

For document search and information retrieval, TF-IDF was developed. It operates by escalating according to the frequency with which a word appears in a document but is counterbalanced by the number of documents in which it appears. Therefore, even though they may seem frequently, words like this, what, and if rank low because they aren't crucial to that document.

TF-IDF (Robertson, 2004) is calculated for each word in a document by multiplying two separate metrics: The number of times a word appears in a document. The simplest method of determining this frequency is to count the number of times a word appears in a document. The length of a document or the frequency of the expression that appears the most frequently in a document is another way to modify frequency. The next metric is inverse document frequency. This statistic shows how frequent or uncommon a word is over the entire corpus of documents. A word is more common the nearer it is to 0. This metric can be obtained by taking the total number of documents, dividing it by the total number of documents containing a word, and then computing the logarithm. This number will be close to zero if the word is widely used and appears in numerous papers. If not, it will go close to 1. The result of multiplying these two factors is the word's TF-IDF score in a document. As the word frequency increases, the value of the word decreases.

2.2.1.3 Word Embedding

Before the discovery of word embedding (Arora et al., 2016; Y. Li & Yang, 2018), the context and semantics of the words were not taken accountable. A natural language model needs to understand the similarities and relations between the words. To understand this, let's consider two sequences "My lunch this afternoon was yummy" and "Lunch I had this afternoon was delicious". Here as we can see, the two sentences have the same meaning. Now, if we try to apply TF-IDF or Bag-of-Words model, treat "delicious" and "yummy" as words that don't mean the same. But as a human, we know that's not the case. To overcome this, we have word embeddings. Word embedding is one of the most common ways to express document vocabulary. It can identify a word's position in a document, its semantic and syntactic similarities, its relationship to other words, etc. Recently we had various text embeddings techniques like word2vec and glove, to name a few.

Word2Vec (CHURCH, 2017; Neelakantan et al., 2015) uses the cosine similarity metric (B. Li & Han, 2013; Nguyen & Bai, 2011) to identify word similarities. The cosine angle of 1 indicates that words overlap or are similar. Words are independent or have no contextual relationship if the cosine angle is 90. It gives similar words identical vector representations. In word embedding techniques, we use cosine similarities to find the relationship between the words. If the angle between two words is 90 degrees, there is no relation between the words. If the angle between words is 0, then words are very similar—graphical implementation of cosine similarities as shown in figure 2.7. Two methods to find word embeddings using word2vec: Skip Gram (W. Cheng et al., 2006) and CBOW (Kenter et al., 2016; Liu, 2020). Using different words as input, the neural network model in CBOW predicts that the target word is highly related to the input words' context. On the other hand, the Skip-gram architecture uses one word as input and predicts all the closely associated context words. While Skip Gram can effectively represent uncommon words, CBOW is rapid and finds a better vector representation for frequent words.

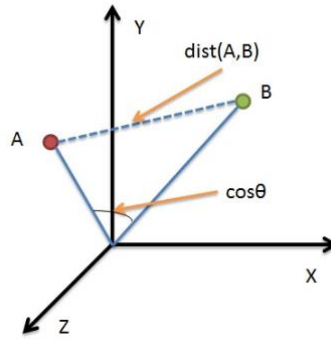


Figure 2. 7: Cousin Similarities Between Two Words (Kai, n.d.)

2.2.1.4 Recurrent Neural Network

A feedforward neural network with internal memory is a recurrent neural network (Ghatak, 2019; Sherstinsky, 2020). The result of the current input depends on the previous computation, making RNNs recurrent because they carry out the same function for every data input. The output is created, copied, and delivered to the recurrent network. It considers both the current input token and the output token it has learned from the prior input when making a decision. RNNs can process input sequences using their internal state (memory), in contrast to feedforward neural networks. Figure 2.8 shows the graphical representation of RNN.

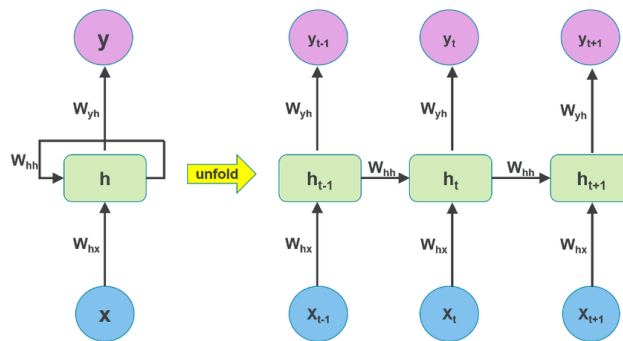


Figure 2. 8: Graphical Structure of RNN (Jian Zheng et al., 2017)

Recurrent networks are distinguished by the fact that each layer of the network uses the same weights. Recurrent neural networks share the same weight parameter inside each layer of the network, in contrast to feedforward networks, which have distinct weights across each node. However, to support reinforcement learning, these weights are still modified using the techniques of backpropagation and gradient descent. Recurrent neural networks use the backpropagation through time (BPTT) (de Jese & Hagan, 2002) algorithm, which differs slightly from conventional backpropagation because it is tailored to sequence data to find the gradients. In classical backpropagation, the model trains itself by computing errors from its output layer to its input layer. This is how backpropagation through time works as well.

But in contrast to feedforward networks, which do not share the same parameters between layers, BPTT adds errors at each time step, which is how it varies from the conventional technique. RNN works best for a short sequence. As the length of the sequence increases, the size of the weight metrics increases. This can result in either exploding gradient or a vanishing gradient.

2.2.1.5 Long Short-Term Memory Network

Long Short-Term Memory Networks (LSTM) (J. Cheng et al., 2016) is a modified version of RNN capable of handling sequences of long lengths using gates that regulate the network's information flow. Although the working of RNN and LSTM is very similar, LSTMs contrast with RNNs and employ gates to choose whether to preserve or discard input.

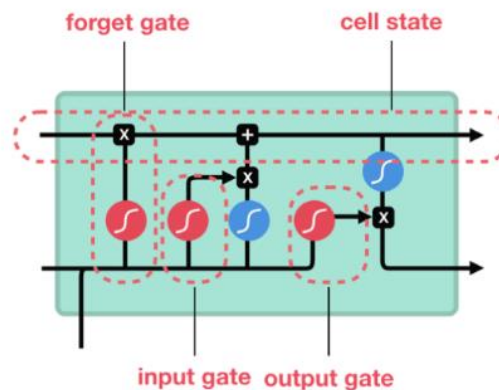


Figure 2. 9: Structure of LSTM Cell (Etqad Khan, 2020)

We can see the standard structure of the LSTM cell in figure 2.9. LSTM networks have four essential components, “Forget Gate”, “Input Gate”, “Output Gate”, and “Cell State” (Graves, 2012). The forget gate selects the data that should be retained and the data that should be destroyed; next, we have the input gate, which updates the state of the cell; the output gate has the information of the previous inputs, it is responsible for deciding what would be the next hidden state. Finally, the cell state traverses the context information from one cell to the other.

LSTM works by passing the previous hidden state, and the current input through a forget gate. As forget gate has a sigmoid activation function; it squashes the values between zero and one. If the resulting value is closer to zero, we discard the values from memory and retain them otherwise. The previous hidden state and the current input are then passed through an input gate that has a sigmoid and tanh activation function in parallel. Here we obtain two values which are then multiplied to get a single output ‘i’. To obtain the next “cell state”, we calculate the product between the output of “forget gate” and the single output ‘i’ of “input gate”.

2.2.1.6 Transformers

Transformers (Vaswani et al., 2018.) are the latest addition to the deep learning models. Transformers like LSTM and RNN are best suited for dealing with sequential data. The major drawback of LSTM and RNN was that as the length of the sequence increases, the model performance seems to degrade. Another disadvantage of LSTM and RNN is that we cannot achieve transfer learning or sharing of weights to fine-tune our model. We can process the entire sequence simultaneously with transformers, thus introducing parallelism. As a result, it takes less time to train the model.

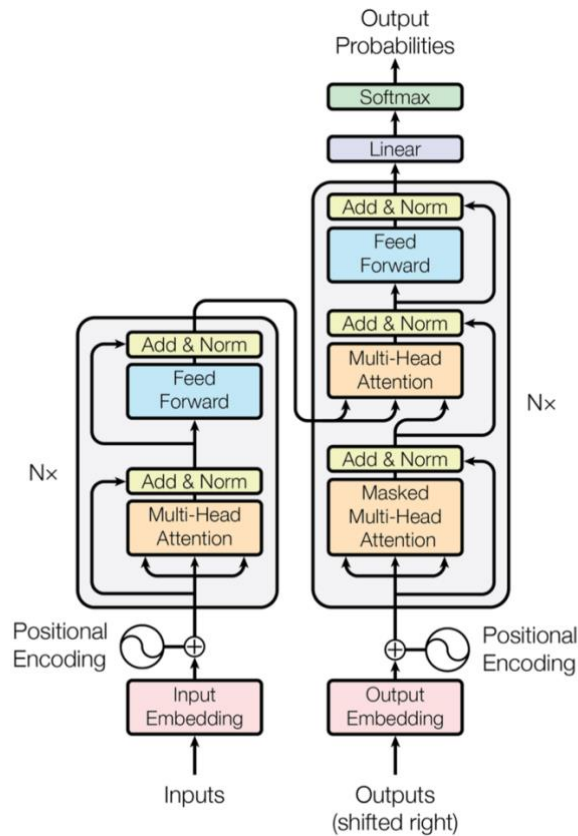


Figure 2. 10: Architecture of Transformers (Vaswani et al., 2018.)

As we see in figure 2.10, the architecture of transformers. Here on the left side, we have a single layer encoder, and, on the right, we do decoder. On a high level, the encoder converts a series of input data into an abstract context vector containing all the information previously learned from that input. Then, using that context vector and the previous output as input, the decoder gradually produces a single output. Let us see the working of transformers in detail.

2.2.1.7 Transformer Encoder:

Encoders consist of three subcomponents, i.e., 'embedding layer', 'positional encoding', and 'encoder stack'.

I. Embedding Layer

Input is fed into a word embedding layer (Denis Rothman, 2021a) as the initial step. One way to visualise a word embedding layer is as a lookup table that can be used to retrieve a learnt vector representation of each word. Since neural networks know through numbers, each word is represented by a vector with a continuous value called embedding dimension.

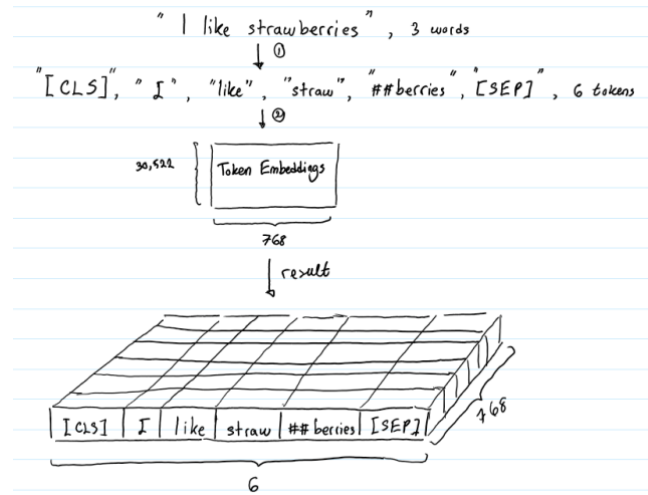


Figure 2. 11: Word Embeddings

We see from figure 2.11 that the sentence is being embedded into a unique representation that has a dimension of 768. The length of the sequence is 6. Sentence length plays a key role when we are using a pre-trained model.

II. Positional Encoding

The embeddings must then be given positional information. We must include some positional information in the input embeddings because the transformer encoder lacks recurrence; unlike recurrent neural networks, thus positional encoding (Denis Rothman, 2021b) is applied. There are multiple ways to add position embedding to our model. Still, the one used in the original transformers multiplied the even and odd index of the encoder output with sin and cos function, respectively. We can see the function in the equation below.

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

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

Make a vector with the cos function for each odd index in the input vector. Use the sin function to build a vector for each even index. Then, merge those vectors into the input embedding that corresponds to them. The model is successfully informed of each vector's position thanks to this.

III. Encoder Stack

The encoder layer must map all input sequences into a context vector to store the context information for each input sequence. It has two submodules, multi-headed attention, and a fully connected neural network. Self-attention is a particular attention mechanism used by multi-headed attention in the encoder. The models can link each word in the input to other words thanks to self-attention. Here a model can associate stop words with other words, thus understanding and generating a natural language-like sequence. As we can see from the structure of self-attention in figure 2.12 (a), there are three inputs, i.e., 'Query', 'Key' and 'Value' vectors. (Denis Rothman, 2021c)

Let's relate this to real-world situations to better understand query, key, and value. Consider searching for a video on YouTube. The item to be searched can be correlated to "query", the various possible values are "keys", and finally, the resultant list of videos can be related to 'values'. Here we first pass the three parameters through a linear layer. The next stage is generating attention scores, as shown in figure 2.12 (b). Here, matrix multiplication (dot product) is performed between query and key, giving a scoring matrix. The scoring matrix determines how much emphasis a word should place on other words. Each word will be assigned a score related to other words in the time step. The more attention is paid, the higher the score. The keys are mapped to the queries in this manner. Once the scores are obtained, we scale them using a SoftMax activation function. This is done to stabilise the gradients. The output vector is then received by taking the attention weights and multiplying them by your value vector. The importance of the words the model learns will continue to increase with more SoftMax scores. The lower scores will overpower the irrelevant words. A linear layer then processes the output from that.

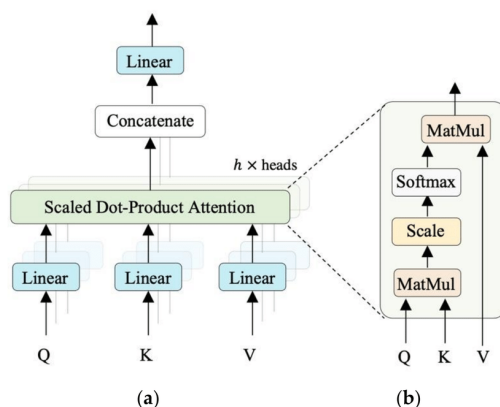


Figure 2. 12: Transformer Self-Attention Mechanism (Vaswani et al., n.d.)

Before applying self-attention, we must divide the query, key, and value into N vectors to turn this computation into a multi-headed attention computation. After then, each split vector goes through the self-attention process. Each self-attention process is referred to as a 'head'. Before passing through the final linear layer, each head's output vector is concatenated into a single vector. The encoder approach would have higher representational power because each head would theoretically learn something different. (Denis Rothman, 2021c)

In summary, multi-headed attention is a transformer network module that computes the input's attention weights and generates an output vector containing encoded data on how each word should pay attention to every other word in the sequence. The output of the multi-head attention block is then pass-through linear and normalisation layer. This concludes one encoder stack. Like this, we can have n number of encoder stacks, such that each layer has an opportunity to learn about the features. Encoders generate context vectors, like the decoder's lookup table, while generating text. Let us now move to the decoder side of the transformers (Denis Rothman, 2021c).

IV. Decoder Stack

The decoder side of the Transformers works like that of the encoder with minor changes. The task of an encoder is to understand the context and form the relation;

the decoder does the opposite job, designed to generate the text sequence. Decoders have two multi-head attention layers followed by feedforward and normalisation layers.

Although the decoder has two multi-head attention layers, they work differently. In the first multi-head attention layer, we have the attention that prevents the model from considering the future prediction into its conditional probability. I am trying to say that as the model is auto-regressive, it predicts one word of a sequence at a time and keeps predicting the values till the end token. Here transformers use conditional probabilities to predict the following sequence. We use the attention mask to prevent the model from looking into the future, as shown in figure 2.13. We then add the scores and the attention mask. Once we have found the sum, we pass it through a SoftMax activation function. This makes the data between zero and one. All the values with -inf will be transformed to zero (Denis Rothman, 2021d) .

0	-inf	-inf	-inf
0	0	-inf	-inf
0	0	0	-inf
0	0	0	0

Figure 2. 13: Don't Look Ahead Mask (Michael Phi, 2020)

The second attention mask works like that of an encoder one. Here key and the query and key vectors are used from the final encoder side of the transformer, which is illustrated in figure 2.14. The final layer of the decoder is a traditional feedforward layer. To the output of the liner layer, we add the SoftMax activation function, which gives us the probability/ max likelihood of that word being selected. The output dimension of the SoftMax will be a vector like the size of vocabulary, i.e., (1, Vocab Size). (Denis Rothman, 2021d)

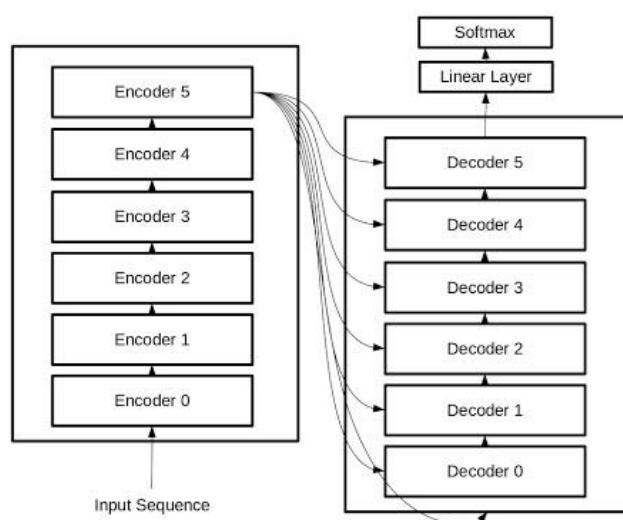


Figure 2. 14: Sharing Key and Query Values Between All the Attention Layers of Decoder(Mandar Deshpande, 2020)

2.2.2 Language Model Development Life Cycle

Language Modelling refers to training a model to learn the likelihood of occurrence of the next word or character in a document based on the previous sequence. It is performed using various statistical and probabilistic techniques. The language model can be trained on a wide range of tasks like text generation, text classification and text summarisation, etc. Because language is incredibly complicated and constantly changing, more complex language models perform better at NLP tasks. Language modelling is the main reason machines can understand quantitative information, as the information is converted from quantitative to qualitative using various language models. Using this, people can communicate with devices as they communicate with others, only up to a limited extent.

2.2.2.1 Data Acquisition

Data acquisition is the process of obtaining signals that reflect the state of the natural world and then transforming those signals into numerical form so that the machine can modify them further. It is necessary to gather the data from numerous sources to store, clean, pre-process, and use it for subsequent mechanisms. Text Pre-Processing. The process of Data Acquisition involves,

Data Discovery: Data discovery is the initial method of data acquisition. It is a crucial step when indexing, distributing, and looking for new datasets on the web and incorporating data lakes. Searching and sharing are the two parts that make up the process. The data must first be tagged, indexed, and then released for sharing via one of the many available collaborative platforms. Most of the time, the data is available in Excel and Comma Separated File (CSV) format, but many websites with data as per our need do not offer the data in the convenient form. This is where we can collect the data using a method known as web scraping. Web Scraping helps collect data from webpages and store it in different formats like Excel, CSV, etc. Web Scrapers finds the correct data over the Internet and take a series of actions to extract the text. The process of web scraping involves fetching and removing. Web Crawler is an essential component of web scraping for conveying the web page for processing later. Once the page is brought, the process of extraction takes place. Here the contents of the web page are parsed, searched, and reformatted, and the data is copied into a spreadsheet.

Data Augmentation: Data augmentation is the following strategy for data gathering. In this context of data collection, we are simply enriching the existing data by adding new external data, which is defined as making something more outstanding by adding to it. Pre-trained models and embeddings are frequently used in deep and machine learning to enhance the number of features to train on.

Data Generation: Creating the datasets manually or automatically is an alternative if we don't have enough data or if any other data isn't available. People are given tasks to gather the necessary data to create the generated dataset, which is the typical technique for manual data building. It is also possible to create synthetic datasets using automatic methods. In cases where data is already there but needs missing values imputed, the data generation approach can also be considered data augmentation.

2.2.2.2 Text Pre-Processing

It is critical to analyse data once it has been collected before using it for analysis or prediction. Text pre-processing is used to prepare text data for model building.

The various steps involved in text pre-processing are:

Tokenisation: It splits the text into smaller units. Here tokens can be words, characters, or subwords. There are two different types of tokenisation techniques, namely,

1. Word Tokenization:

This is the most used tokenisation algorithm. The text is divided into separate words using specific delimiters, and based on these delimiters, different word-level tokens are formed. Word tokenisation uses pre-trained word embeddings such as Word2Vec and GloVe.

2. Character Tokenisation:

It splits a piece of text into a set of characters. By using this tokeniser, memory and time complexity can be reduced. In character-based tokenisation, reducing the vocabulary size comes at the expense of increasing the sequence length.

3. Unigram Tokenisation:

Unigram tokenisation begins with determining the necessary vocabulary size. The primary difference between unigram and the other two ways is that we do not start with a limited vocabulary of characters. Instead, all the words and symbols are found in the base vocabulary. To get at the final vocabulary, tokens are gradually deleted. The most common unigram tokeniser is Sentence Piece, which takes the entire string as input in its raw format and then uses unigram tokenisation or byte pair encoding tokeniser.

4. Sub-word Tokenization:

It splits a piece of text into sub-words or n-gram characters. The three main sub-word tokenisers used are:

1. Byte Pair Encoding (BPE): Among transformer-based models, Byte Pair Encoding is the most often used tokenisation approach. It is a word segmentation method that iteratively merges the most frequently occurring character or character sequence.
2. Word Piece: Word Piece is like Byte Pair Encoding, which first includes all of the characters and symbols into its base vocabulary. The vocabulary size is set here, and subwords are added until the limit is reached. It selects the one with the highest likelihood of training data. This indicates that the model is trained on the base vocabulary before choosing the pair with the highest probability.

Lower casing is the most used pre-processing step in NLP, where all the characters are converted to lower case. But most of the time, this pre-processing step leads to the loss of information.

Stop word removal: Stop words are words in any language that add little meaning to a sentence. Stop words can be removed because they do not add value to our analysis.

Stemming: Stemming refers to removing the suffix from a word and reducing it to just its root word. There are two error measurements in Stemming Algorithms:

- i. Under Stemming occurs when two inflected words should be stemmed to the same root word but are not. This is also known as a false negative.
- ii. Over Stemming happens when two inflected words should not have stemmed from the same root word. This is also referred to as a false positive.

Lemmatisation: The process of replacing a word with its root or headword, known as lemmatisation. A lemmatise employs a database of word synonyms and word ends to verify that only comments that imply the same thing are combined into a single token.

2.2.2.3 Model Development

In the next stage, we develop our Transformer based deep learning model. Depending on the model, this process can be as simple as calling the object of the model and loading the weights or something like an encoder-decoder model where the encoder takes in the input data. The decoder takes the input from the encoder and processes it. A combination of the encoder-decoder model is called sequence-to-sequence modelling. Here we have used the pre-trained model from the “Huggingface” library. Here to make the training and evaluation steps more accessible, I have used the “Pytorch Lightning” framework that allows me to automate some functionalities. A detailed explanation of model development and architecture will be in chapter 4

2.2.2.4 Model Evaluation

After developing language models, it is assigned to perform a particular NLP task. After completing the task, the model is evaluated based on the Perplexity and Shannon Visualisation Method. In the Perplexity Method, when normalised by the number of words in the test set, perplexity is the multiplicative inverse of the probability of the language model assigned to the test set. If a language model can forecast words not included in the test set, The Shannon Visualisation Method uses the trained language model to produce sentences.

2.2.2.5 Deployment

Deployment is a curtail step to use the model in real life. With recent advancements, we can deploy the model using Fast API. After our model has been successfully trained, we save the model's weights. These weights can be transferred from one machine to another with ease. Deployment of the deep learning model can be a bit challenging as the model are enormous. This also means that it will take time for the model to predict and return the values to the end-user / client.

2.2.3 Application Of Language Modelling

Language Models have a wide range of applications in Natural Language Processing Tasks such as,

Speech Recognition – Speech Recognition applications such as Google Assistant, Alexa, Siri etc., use language models as one of their major components. Here the system is provided with a sequence of words used to predict the probability that the given term is next.

Sentiment Analysis - Finding the general orientation of a text, whether positive, negative, or neutral, is the primary objective of sentiment analysis. Language models have recently produced excellent achievements in improving the English text classification accuracy.

Machine Translation - Phrase-Based Machine Translation relies on a decoding algorithm which tries to cover the original sentence into a phrase. The language-based model determines the state score in the decoding phase.

Chapter 3

3 Problem Specification

As specified in the Introduction here, we will be developing a Transformer based model which can generate comments for various programming languages like Python and JavaScript. Chapter 2 summarises multiple methods and techniques I can use while developing the model. To get a better understanding of the problem statement, I also researched other topics that are closely related to it. Here we are dealing with code-to-text. But similar problem statements like translating from one programming language to another, i.e., code-to-code translation and given text, we were to generate code that will be a text-to-code generation task. I have tried experimenting with code-to-text generation using RNN, LSTM and various pretrained transformer models.

3.1 Code Comment Generation

Artificial intelligence is widely used to create models that aid programmers in writing better source code. The usage of various AI models, such as code autocompletion, autosuggestions, unit test aid, bug identification, code summarising, etc., helps developers do their work more quickly. Code documentation is crucial when completing the project. It might be challenging for future developers, testers, or clients to grasp the operation of substantial code snippets with hundreds of lines of code; thus, code documentation in the form of comments is crucial. Most developers find it challenging or forget to add comments or documentation. Looking at the importance of the words in our day-to-day coding activities, I have come up with a solution to overcome the issues of comment generation. So far, we have models that generate comments specific to the programming language it was trained on. But as the computer industry proliferates every time, we have a new language. We are supposed to develop a model which generates appropriate comments. In this project, we have tried to develop a model that can create words for the language it was trained on and for a new language with a similar structure.

3.2 Use of Pretrained Models

Although the data set size is small for the NLP models, the amount of time and the compute required to train the model is substantial. Thus, introducing a model from scratch would be challenging. Language models are also known for either having vanishing gradients or exploding gradients. Also, as the length of the sequence increase, the effectiveness of the model in creating a good prediction reduces. Thus, to solve all this, we will be working on transformers. As discussed in chapter 2, we can use a pre-trained model with transformers, thus reducing the time taken to train the model. Some pre-trained code-text translation models are 'Code BERT', 'Code T5', and 'Roberta (code)'. From the initial research, I have decided to go with Code BERT. As we know, BERT is an encoder-only model. The encoder's job is to understand and

generate the context of the sequence. For the decoder, we have used the RoBERTa model.

Chapter 4

4 Model Design

In this section, we will be talking in detail about the working of our project, more specifically, how our model works. At the core, this model is a sequence-to-sequence model (Sutskever et al., 2014; Tang et al., 2016). This means we have an encoder that takes in input from the data loader and gives out a context vector. The output of the encoder becomes the input to the decoder. The decoder makes use of the context vector and generates an output sequence. As I have mentioned in chapter 2, after feeding our model with the data, our model processes it in a forward direction and then computes the loss. Once we have calculated the loss, we update the gradients by backpropagation. Learning rate also plays a crucial role in how quick we can optimise the weights. As we see here are multiple steps involved in a single process. We used the “Pytorch Lightning” framework to make our code clean. This way, we create the hassle of manually calling for backpropagation and optimisation. Instead, we only define a function, and the lightning framework manages the data loading and training iterations in its back end. Figure 4.1 represents the training loop. We define functions that handle the forward propagation, computing the loss and text generation. Here as we are using a pre-trained model, we clone the weights of our encoder to the generator to ensure that we have similar consequences at the generator.

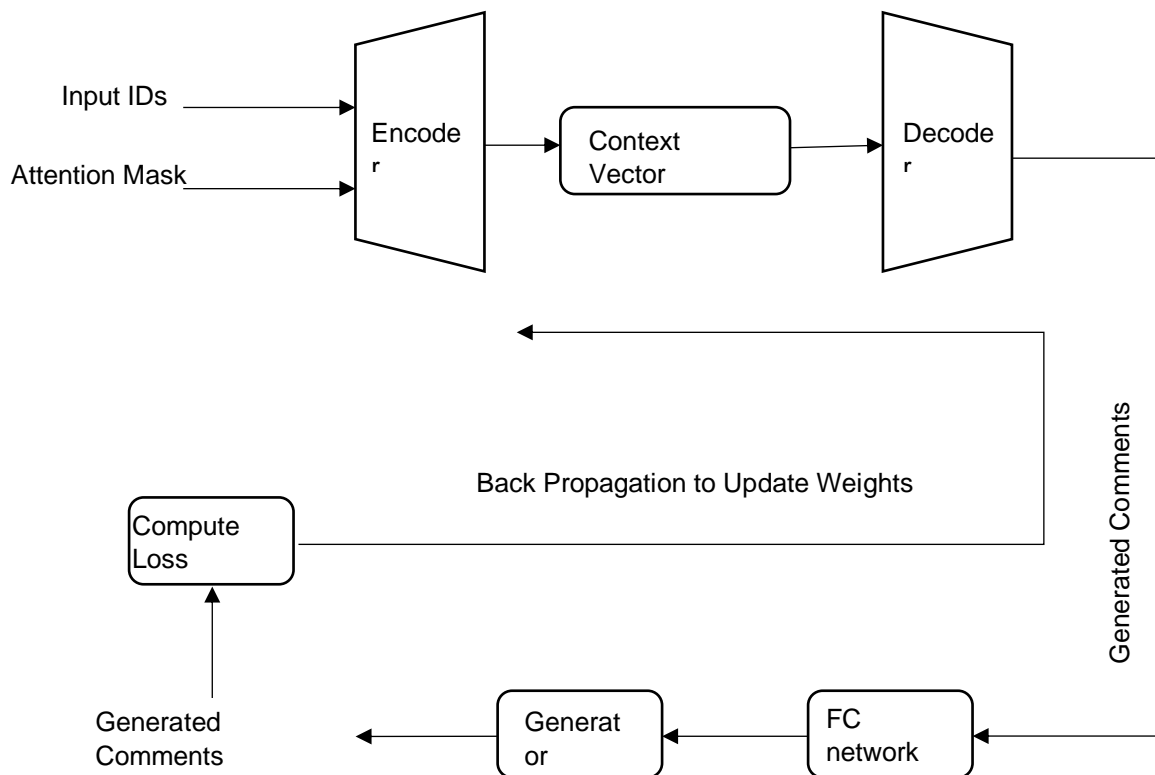


Figure 4. 1: Training Loop

4.1 Forward Propagation

This is a common ground for both when a model is training and during evaluation. We define this function as “forward()” with input ids and attention mask of both the source and target and finally initialise them to “None”. During the training stage, we provide all the four values to the arguments, but during the validation or test stage, we pass only the source attention mask and input ids. We can see the definition of the function in figure 4.2 below.

```
def forward(self, src = None, src_key_padding_mask= None, tgt= None,
```

Figure 4. 2: Function Definition

We provide source ids and an attention mask to generate the context vector to the encoder. We will only consider the output from the last encoder layer to get the context vector. The shape of the data will be “*source length, batch size, embedding dimension*”. In the next stage if our model is in the training stage, we will first encode the target ids using pre-trained encoder of Code BERT. As discussed earlier, to have a self-attention mechanism in the decoder we will have to use “don’t look ahead mask” as discussed in chapter 2. This mask is usually a square matrix of the size that is equal to the target length. In the final stage we pass the data through the decoder and the output shape of the decoder is “*target length, batch size, embedding dimension*”. We can see the implementation of both encoder and decoder below.

```
# encoder output, this gives us the context vector which is present in the last decoder layer
encoder_output = self.encoder(src, attention_mask = src_key_padding_mask )[0]-> torch.Size([32, 212, 768]) -> [N,S,E]
encoder_output = encoder_output.permute([1,0,2]).contiguous() # rearranging tensor to shape -> torch.Size([212,32,768]) -
> [S,N,E]

tgt_mask = -1e4 *(1-self.bias[:tgt.shape[1],:tgt.shape[1]]).to(device= device) # shape -> trg_mask > torch.Size([42, 42])

# encode the target embeddings
tgt_embedding = self.encoder.embeddings(tgt) # shape: torch.Size([32, 42, 768]) -> [N, T, E]
tgt_embedding = tgt_embedding.permute([1,0,2]).contiguous() # shape: torch.Size([42, 32, 768]) -> [T, N, E]

# memory key padding mask
memory_key_padding_mask = (1-src_key_padding_mask).bool() # shape : torch.Size([32, 212]) -> [N, E]

# output of the decoder
output = self.decoder(tgt = tgt_embedding,
                      memory = encoder_output,
                      tgt_mask = tgt_mask,
                      memory_key_padding_mask = memory_key_padding_mask ) # torch.Size([42, 32, 768]) -> T, N,
E
```

Figure 4. 3: Encoder and Decoder of definition

Now, if the data were in an evaluation or test mode, we would only pass the source ids and attention mask. The working of this is precisely identical to that of an encoder, but

the only difference is that we iterate the source length number of times to generate the comments for the input code. The search algorithm that we will be using here is a Beam search. As the decoder is autoregressive, it generates sequences word by word. It will stop the loop as it generates the end of sentence token.

4.2 Search Mechanism

The output decoders give the probabilities of the character being part of the sequence. As the size of this is equal to the size of the vocabulary. As we must search who has the maximum probability, it would cause delays in producing the output, and only one value is selected. Now to enhance this, we use the modified version of the greedy search algorithm, Beam Search (Bausch et al., 2021; Meister et al., 2020; Xu & Todorovic, 2016; Yang et al., 2019). Here we choose n number of possibilities for the sequence. n is a hyperparameter called beam width. This works because the model compares how well the n number of words go along with the previously selected words. This way, we get the optimal solution for generating a good sequence.

4.3 Loss Function and Optimiser

The objective function used here is a Cross-Entropy loss. Cross-Entropy loss measures the similarity between the two vectors. The probability of each predicted word is compared against the likelihood of the actual class. To change the model weights during training, the cross-entropy loss is used. The goal is to reduce loss; hence, the better the model, the smaller the loss. We define cross-entropy loss as shown in the equation below. 'X' is an accurate label and 'y' is a predicted label. As the loss reduces, closer is the predicted value to the original data.

$$Loss = - \sum_{i=1}^n X_i \log(y_i)$$

The type of optimiser that we will be using is "AdamW" (Llugsi et al., 2021; Zhang, 2018). Using this, the change in learning rate does not change the average weighted decay. A vital component of an optimiser is the learning rate. If the learning rate is too low, we might never reach the global minima; if the learning rate is tremendous, we might overshoot the global minima point. Having a static learning rate is also problematic. Thus, we have used a learning rate scheduler to change the learning rate accordingly. We clip the gradients as they reach one to prevent the exploding gradient. Figure 4.4 illustrates the code implementation of the optimiser and scheduler.

```

# Prepare optimiser and schedule (linear warmup and decay)
no_decay = ['bias', 'LayerNorm.weight']

optimizer_grouped_parameters = [
    {'params': [p for n, p in self.named_parameters() if not any(nd in n for nd in no_decay)],
      'weight_decay': 0.0 },
    {'params': [p for n, p in self.named_parameters() if any(nd in n for nd in no_decay)],
      'weight_decay': 0.0}
]

optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr = self.lr , eps= 1e-8 )

nn.utils.clip_grad_norm_(self.parameters(), 1.0)

torch.autograd.set_detect_anomaly(True)

lr_scheduler = {
    'scheduler': transformers.get_linear_schedule_with_warmup(optimizer= optimizer,
num_warmup_steps = 5000, num_training_steps= self.epoch * len(Test) ),
    'interval' : "step",
    'frequency' : 1,
    'monitor' : "validation_loss",
}

```

Figure 4. 4: Defining our *optimiser* and scheduler

4.4 Evaluation of Results

The goal of evaluating text is to have an unbiased judgement about the quality of the generated comments. Summarising a text's primary point can help you understand it better and will help you evaluate it. Then, to completely comprehend the text, we might start to study various aspects of it. Finally, we can use the knowledge gained from the evaluation matrix to create our own opinions about the model. There are multiple ways to evaluate the text. Here we will use BLEU score and Cross encoder to see the quality of the comments generated.

4.4.1 BLEU score

While dealing with text, one popular approach to evaluating the sentence with a reference is the BLEU score (Chatoui & Ata, 2021; Datta et al., 2021; Malik & Baghel, 2016). BLEU is an acronym for Bilingual Evaluation Understudy Score. The BLEU score ranges from 0 to 1 and measures the similarity between the generated and reference comments. When the BLEU score is zero, what that means is the two sets of sentences don't overlap, and when both the sentences are identical and overlap, the blue score is 1. In this project, we will use NLTK built-in function to generate the BLEU score to implement the BLEU score.

4.4.2 Similarity Score

The drawback of the blue score is that it fails to understand the sentence. BLEU score gives more weightage to the word of the sentence over the actual meaning of the

sentence. To see how similar two sentences are, we use Cross Encoder (Choi et al., 2021; Devlin et al., 2018; Malik & Baghel, 2016; Mrinalini et al., 2022). We have this by having a classification head to a BERT model. A pair of text documents are entered into the BERT-based cross-encoder model, which then outputs the likelihood that the two texts are similar. Implementation of BERT based cross encoder is illustrated in figure 4.5. In this project we use “sentence_transformers” library to use cross encoder.

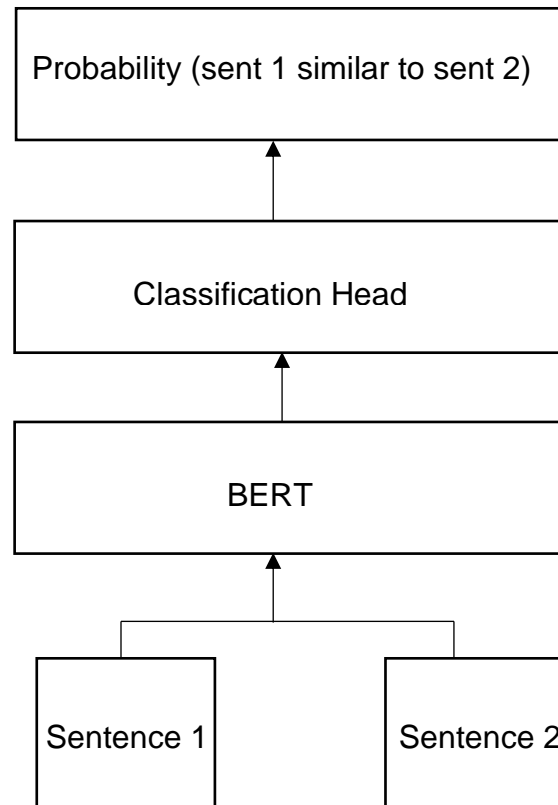


Figure 4. 5: BERT Based Cross Encoder

Chapter 5

5 Implementation Details

This section discusses the model's implementation, development, and training process, which are explained in chapter 7. As we are trying to develop a language model, the generic steps are described in section 2.2.2 of chapter 2. Keeping that structure in mind, we will discuss this project's details.

5.1 Dataset

We will require function (methods) and docstring pairs of various languages to train our model. The input to the model will be functions (methods). To validate if the model has generated the correct sequence and compute the loss for fine-tuning the weights, we will need a docstring that describes the function (methods); thus, this is a dependent feature. To obtain code-docstring pair, we can scrape through the official documentation of that programming language. I have used a publicly available dataset by “Coresearcher” for simplicity. “Coresearcher” provides data for various problem statements, but for our task, which is a code-to-text, i.e., code comment generator dataset available for languages like Python, Java, PHP, JavaScript, Ruby, and Go. Here we will be using the dataset for Python and JavaScript. Figure 5.1 below illustrates a pair of codes and their respective docstring. Here we are provided with 900,000 records for Python language and about half a million for JavaScript.

code	docstring
<pre>def _source_for_file(self, filename): """Return the source file for 'filename'.""" if not filename.endswith(".py"): if filename[-4:-1] == ".py": filename = filename[:-1] elif filename.endswith("\$py.class"): # jython filename = filename[:-9] + ".py" return filename</pre>	<pre>Return the source file for 'filename'.</pre>

Figure 5. 1: Preview of Dataset

This data can be loaded in multiple ways. This is accessible by the AWS s3 bucket, or we can download this using the Huggingface library. The implementation details for loading the dataset can be found in the [appendix](#). As we know, using the raw data to train our model is impossible as this can lead to inaccurate predictions or exceptions. In the next section of this chapter, let's see the various data pre-processing techniques we have used to clean the data set.

5.2 Data Pre-Processing

As we see in figure 5.1, the code side of the data set has the docstring within the multi-line comments; also, as the structure of the code, i.e., indent, is not taken into consideration, we must remove extra spaces and the new line. We have various other changes that need to be made. Let's see the different data pre-processing steps we have implemented.

1. Removing Docstring from The Code:

Here, we must remove the docstring from the code to prevent the model from overfitting. Also, I have trained the model with and without the presence of the docstring/ comment in the code. We obtained such a result that the training loss was shallow, and the validation and test loss were higher. This is a clear sign that it is overfitting. The detailed code implementation of the model is present in the [appendix](#).

2. HTML Tag Removal:

As this dataset is originally scrapped from the official documentation of the programming languages, that are high possibilities of having HTML Tags in both code and docstring. The presence of a docstring is like having a noise. Failing to remove can lead to poorer performance on unseen data as it is not common to have HTML tags on a programming language. The detailed code implementation of the model is present in the appendix section 9.3.1.

3. Remove Duplicate Values:

We must eliminate the duplicate values as this will only make our model biased towards that feature. While training a model, be it numeric or a text, it is essential that we have variance in the data set as this will prevent the model from overfitting and being biased towards specific patterns or features. Here we have used Pandas' built-in function to eliminate the duplicate values

4. Handling missing features:

As the dataset size is huge, i.e., 900,000 records, it is no surprise that some data will be missing. If we train a model with a missing code of a doc string, adds noise to the final model, or an exception will prevent you from training the model. Here, we will iterate over every record to handle missing data and see if that cell is empty/ missing. If yes, the column number is counted. Finally, we will only select those rows that have no missing values.

5. Handle Outliers:

As we have discussed earlier, Transformers were developed to handle a text of any length. But as the sequence length increases, the training also increases. Some of the pre-trained models have lit to the line size, and here, as we are using code BERT, the sequence length here will be 512. Outliers in the text data are where a small set of records have a long code size or long comment size. We will add padding to other sequences if we don't handle these outliers. This intern led to an inefficient model.

Here, we have dealt with the outliers by calculating the lengths of all the comments and the code. We achieved this by tokenising it using a BERT-based pre-trained tokeniser and plotting it, as shown in figure 5.2. We can see that most of the code and comments have most of sequence lengths less than 1000. We first eliminated the outliers to get an accurate value by considering the values within the upper and lower quartile. In the next step, we found the average length of the comments and code. Finally, we retained the code and words within their respective mean sizes. The detailed code implementation of this is present in appendix section 9.3.2.

Figure 5. 2: Lengths of code and comments

5.3 Create Input IDs and Attention Mask

We can see the implementation of the `encoder_plus()` function for the single code in figure 5.3

Figure 5. 3: Use of 'enoder_plus()' function to generate 'input_ids' and 'attention_mask'

'attention_mask' to a list. PyTorch accepts only tensors while training the model, so we convert the 'input_ids' and 'attention_mask' arrays to a tensor of type long. The detailed code implementation of this is present in appendix 9.4.

5.4 Pytorch Dataset and Data Loader

The torch dataset class is an abstract class that represents the dataset. Instead of treating the dataset as a collection of data and labels, it enables us to handle it as an object of a class. Every time the Dataset object is called, it yields a combination of [input, label]. With the tensor dataset, we can have multiple values. What I mean by this is that an object of the tensor dataset can have 'code input_ids', 'code attention mask', 'comment input_ids' and 'comment attention mask' all at once. This enables us to a cleaner code. In Pytorch to create dataset we use a built-in class "`torch.utils.data.TensorDataset()`". The arguments for this class are the values you would like to have to train the model.

In the next step, we pass the dataset through the data loader. As we work on large-scale deep learning models, we usually encounter the problem of memory overflow. This occurs when we send a large amount of data to GPU RAM. Now to overcome this issue, we must divide our data into batches. Having a suitable batch size is critically very important. We split the data into batches using Data Loader. In addition to dividing into batches, we can also add a pre-processing function. In Pytorch, to create a dataset we use a built-in class "`torch.utils.data.DataLoader()`". The most used arguments for this class are tensor dataset, batch size, sampler and drop last. The detailed code implementation is present in appendix section 9.4.

5.5 Load Pre-Trained Model

As mentioned earlier, we will use pre-trained models to fine-tune our model. Because of this, we also used tokenisers explicitly developed for this model. Now, as discussed in chapter 4, BERT is an encoder-only Transformer. Now we create an object of our encoder and load the pretrained weights from the "Microsoft/codebert-base". Apart from this, we also load the configurations used by the Code BERT. Configuration can include a number of layers, attention heads and hidden sizes. Now that we have our encoder, we now define our decoder. We will not load weights to the decoder. Instead, we will have a similar configuration used in the encoder. The detailed code implementation is present in appendix section 9.5.

5.6 Setting Up Call-Backs

During a process training of a neural network, a function known as a 'callback' may be called frequently. This function typically serves to validate or correct behaviours. Callbacks can specify what occurs before, during, or after a training period. This is extremely helpful to log progress or halt training if our model has obtained sure accuracy. This system is known as early halting. For instance, if you select a training period of 1000 epochs, but the necessary accuracy is already attained by epoch 200, the learning will end immediately to prevent overfitting your model and to save expensive compute resources. As we are using the Pytorch Lightning module, we have decided to use three built-in callbacks 'EarlyStopping', 'ModelCheckpoint', and

'LearningRateMonitor'. These built-in models are present in "pytorch_lightning.callbacks". Here we will continuously monitor and log the learning rate at every step of our epoch. For the early stopping, we will look into reducing validation loss. We will need to have a difference in a loss greater than 0.001, and if the loss is less than 0.001 for five consecutive times, we will terminate the model training.

Similarly, we will save only that model that has generated the minor validation loss for the model checkpoint. The object of these callbacks is passed as a callback argument to the Trainer class of the Lightning module. Below we see the implementation of callbacks in our project.

```
# lightning configuration

lr_monitor = LearningRateMonitor(logging_interval = "step" )

early_stopping = EarlyStopping(monitor= "validation_loss",
                               min_delta = 0.001,
                               patience= 5,
                               strict= True,
                               verbose= True,
                               mode = 'min')

model_checkpoint =
ModelCheckpoint(dirpath="./saved_javascript_model_weights/",
               save_last=True,
               monitor="validation_loss",
               save_on_train_epoch_end = True,
               mode="min",
               save_top_k = 0)
```

Figure 5. 4: Defining Our Callbacks

5.7 Model Training and Evaluation

The conventional way of training the model is using two for-loops, one that iterates over the number of epochs and the other that iterates over the batch. Once we calculate the loss for every set, do backpropagation and call the function of the optimiser. The problem with this approach is that it will lead to a complex or messy code structure. Using the Pytorch lightning module, we define every training process step as a function. We have a forward function that calculates either the loss from the encoder during the training process or returns a list of words' probabilities. To initiate the training process, we will first create the object of the Trainer class of Pytorch. The implementation of this is in figure 5.5. We logged all the values into weights and bias API to track the model performance in real-time. Not finally, train the model; the object of the trainer class has a built-in function "fit()," which automatically trains the model—implementation of this in figure 5.6. During the evaluation process trainer class automatically puts the model into eval mode and runs the model through validation data.

```
trainer = Trainer(callbacks= [lr_monitor, early_stopping, model_checkpoint],
                    devices= 1 ,
                    gradient_clip_algorithm= 'norm',
                    accelerator='gpu',
                    enable_progress_bar= True,
                    default_root_dir = "./saved_javascript_model_weights/default_root_dir/",
                    max_epochs = epoch ,
                    logger = wandb_logger,
                    strategy="dp" , precision=16)
```

Figure 5. 5: *initialisation* of Trainer class

```
checkpoint = torch.load('/notebooks/saved_python_model_weights/last.ckpt')
model.load_state_dict(checkpoint['state_dict'])

trainer.fit(model)
```

Figure 5. 6 Loading the Pre-Trained Weights and Training the Model

Chapter 6

6 Results and Evaluation

6.1 Model Evaluation

The dataset was enormous, so we had to train this in multiple sessions. The information about the model training and evaluation loss gives us an idea if the model tends to overfit or underfit. We have used weights & bias API to capture the losses in real-time. By logging every step and epoch during our training and validation phase, we obtain the graph as shown in figure 6.1 and figure 6.2 for Python and JavaScript, respectively.

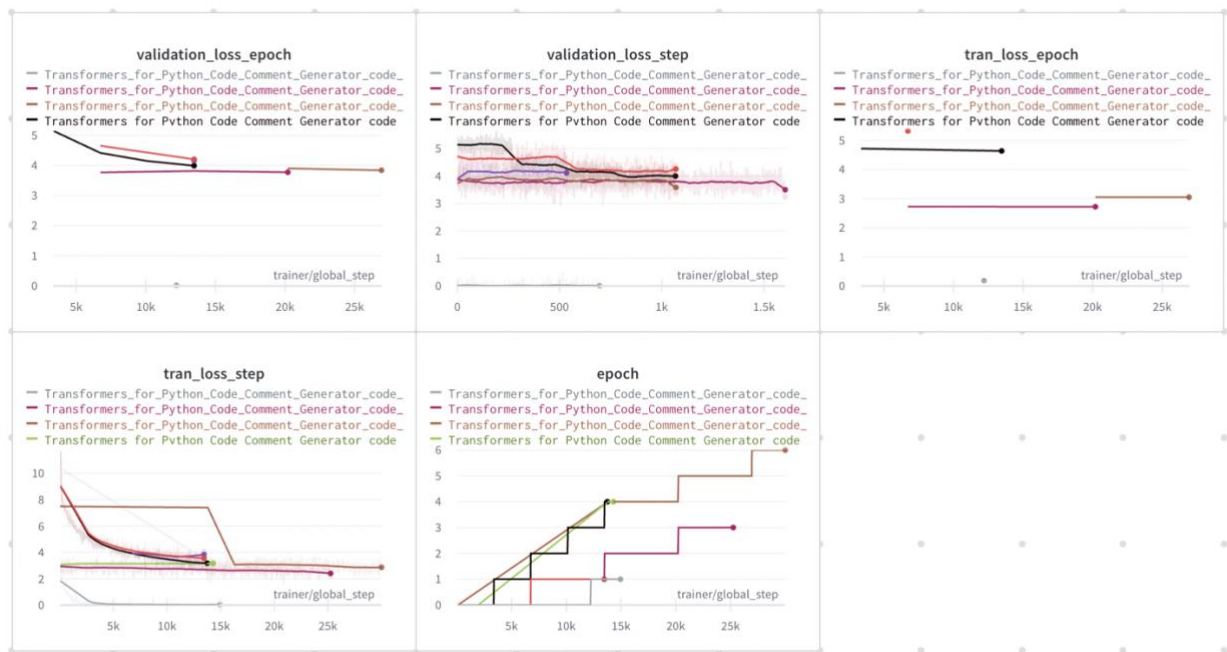


Figure 6. 1: Training and Evaluation Parameters for Python

As we see here, the model does not seem to under fit or overfit as the difference between train loss and validation loss is minimal. For every epoch, there is a gradual improvement in the loss. The learning rate seems to adjust according to the requirements as the model progresses in its training. Time taken per epoch for the python model was approximately 8 hrs, and the total time taken was 15 days. On the other hand, the JavaScript model took around 2 hrs per epoch, and we achieved a satisfactory result by the 6th epoch, which took about 12 hrs to train.

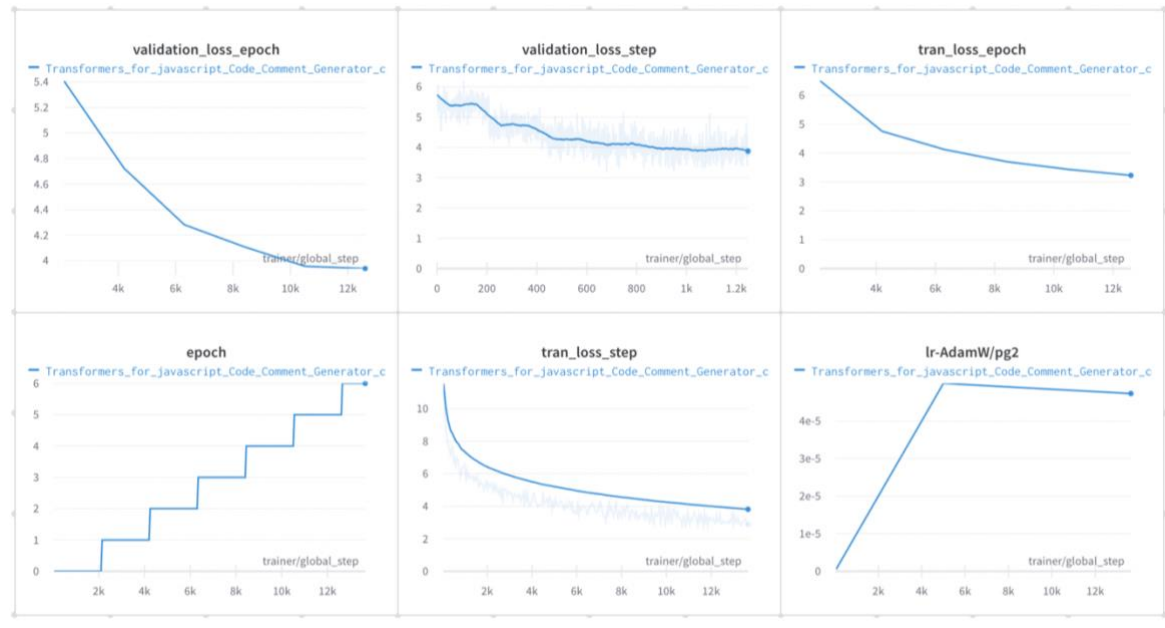


Figure 6. 2: Training and Evaluation Parameters for JavaScript

6.2 Generated Comments

In this section, we will look at some samples of the comments generated during the test phase. We use BLUE score and Cross Encoder to evaluate the generated comments. In order to generate comments for our test cases, I have defined a function that takes code as input and generates comments. Implementation details of the function is in appendix section 9.6

<pre>code : def _fallback_default_verify_paths(self, file_path, dir_path): """ """ for cfile in file_path: if os.path.isfile(cfile): self.load_verify_locations(cfile) break for capath in dir_path: if os.path.isdir(capath): self.load_verify_locations(None, capath) break True comment : Default verify paths are based on the compiled version of OpenSSL. Generated Comment : try to find the default verify paths the blue score for generated comment is 0.29882821494213035 the similarity between sentences is 0.45822611451148987</pre>	<pre>code : def _hide_column(self, column): """ column = _ensure_string_from_expression(column) new_name = self._find_valid_name('__' + column) self._rename(column, new_name) True comment : Hides a column by prefixing the name with '_\' Generated Comment : hide a column the blue score for generated comment is 0.06362676745557341 the similarity between sentences is 0.7088726758956909</pre>
<pre>code : def print_kernel_code(self, output_file=sys.stdout): """ print(self.kernel_code, file=output_file) True comment : Print source code of kernel. Generated Comment : print the kernel code. the blue score for generated comment is 0.5070993931197395 the similarity between sentences is 0.8882415890693665</pre>	<pre>code : def _build_point_formats_dtypes(point_format_dimensions, dimensions_dict): """ """ return { fmt_id: _point_format_to_dtype(point_fmt, dimensions_dict) for fmt_id, point_fmt in point_format_dimensions.items() } True comment : Builds the dict mapping point format id to numpy.dtype Generated Comment : build a dictionary of point formats. the blue score for generated comment is 0.35176163081632544 the similarity between sentences is 0.5469182133674622</pre>

Figure 6. 3 Generated Comments for Python


```

code :
def list_messages(self):
    """
    messages = sorted(self._messages_definitions.values(), key=lambda m: m.msgid)
    for message in messages:
        if not message.may_be_emitted():
            continue
        print(message.format_help(checkerref=False))
    print("")

True comment :
Output full messages list documentation in ReST format.

Generated Comment :
list messages.

the blue score for generated comment is 0.04624620245536924

the similarity between sentences is 0.5436753034591675

```

```

code :
def get_single_axis_values(self, axis, dataset):
    """
    """
    data_index = getattr(self, '%s_data_index' % axis)
    return [p[data_index] for p in dataset['data']]

True comment :
Return all the values for a single axis of the data.

Generated Comment :
return the values of a single axis.

the blue score for generated comment is 0.566139416250068

the similarity between sentences is 0.8609653115272522

```

```

code :
def get_stylesheet_resources(self):
    """
    # allow css to include class variables
    class_vars = class_dict(self)
    loader = functools.partial(
        self.load_resource_stylesheet,
        subs=class_vars)
    sheets = list(map(loader, self.stylesheet_names))
    return sheets

True comment :
Get the stylesheets for this instance

Generated Comment :
get all stylesheet resources

the blue score for generated comment is 0.3698623928739615

the similarity between sentences is 0.7875012755393982

```

Figure 6. 4: Generated Comments for Python

```
code :
function _splitAndTrim(str, seperator, limit){
  return str.split(seperator, limit).map(function(v){
    return v.trim()
  })
}

True comment :
_splitAndTrim: Split string by seperator and trim result

Generated Comment :
split string

the blue score for generated comment is 0.025561533206507392

the similarity between sentences is 0.6802669167518616
```

```
code :
function genPid() {
  if (!Uuid) {
    // module.paths = [Path.join(Path.resolve(CacheDir), 'node_modules')];
    Uuid = require('uuid/v4');
  }
  let str = Uuid();
  let pid = str.replace(/~/g, '').toUpperCase();
  return pid;
}

True comment :
generate a 32 character hexadecimal pid

Generated Comment :
generate pid

the blue score for generated comment is 0.09640774522025064

the similarity between sentences is 0.6796028017997742
```

Figure 6. 5: Comments generated for JavaScript

```

code :
function() {
  set(this, 'isReloading', true);

  var record = this;

  var promiseLabel = "DS: Model#reload of " + this;
  var promise = new Ember.RSVP.Promise(function(resolve){
    record.send('reloadRecord', resolve);
  }, promiseLabel).then(function() {
    record.set('isReloading', false);
    record.set('isError', false);
    return record;
  }, function(reason) {
    record.set('isError', true);
    throw reason;
  }, "DS: Model#reload complete, update flags");

  return DS.PromiseObject.create({ promise: promise });
}

```

True comment :
Reload the record from the adapter.

Generated Comment :
reload a model @return {promise}

the blue score for generated comment is 0.25749092688026104

the similarity between sentences is 0.3237510323524475

Figure 6. 6: Comment Generated for JavaScript

```
code :
function() {
    this._ensureViewIsIntact();
    this._isRendering = true;
    this.resetChildViewContainer();

    this.triggerMethod('before:render', this);

    this._renderTemplate();
    this._renderChildren();

    this._isRendering = false;
    this.isRendered = true;
    this.triggerMethod('render', this);
    return this;
}

True comment :
Renders the model and the collection.

Generated Comment :
render rendered view

the blue score for generated comment is 0.14490095191124455

the similarity between sentences is 0.49453282356262207
```

Figure 6. 7: Comment Generated for JavaScript

To conclude, the model yields satisfactory results for both Python and JavaScript. Although the blue score between the generated comment and actual comment is comparatively less, showing the model generates unique comments, the cross-encoder score between the two comments is satisfactory. Models seem to understand the semantics of the code structure, but the maximum weightage is given to the function's name in most cases. If the model cannot generate a comment, it returns a list representing the input and output types of the function.

Chapter 7

7 Implications and Future Work

7.1 Summary

Successful implementation of the Code BERT model for python and JavaScript languages. The intuition of extracting and understanding the programming language's semantics to generate natural language-like comments has been achieved. The major challenge that was faced during the development phase was the handling of data. If the batch size were set to anything above 32 measures, the system would crash, and if we have a smaller batch size, this will lead to a model being less generic and underfit. Another challenge worth noting was the implementation of the sequence-to-sequence model, which included the concatenation of encoder and decoder to generate the output. Although the generated comment quality doesn't come close to the state-of-the-art model, improvements can be made to get better results. To conclude, we can use Transformers to understand program language and process the output as natural language.

7.2 Limitations and Future Work

The model seems to provide good results when the code length is small to medium. But when the code size increases, the model produces randomly generated garbage values. The model fails to capture the structure of the programming language. What I am trying to say is, let's say, for a language like Python, where indentation is necessary, the model treats the indent as a vast space. Taking this problem as a motivation upon further research, I figured out that the traditional way of dealing with data will not be able to capture the semantics of the structure. A new type of neural network called GNN – Graphical Neural Network can be used to overcome this problem. Here instead of treating code as a text, we treat it as a graph node. i.e., using AST token, we can capture and represent the data as a node, thus preserving the structure of the input code. As this model can understand the structure rather than only text, there are high chances that this model will overcome the drawback of this model.

8 Bibliography

- Alexis Perrier. (n.d.). *Introduction to Natural Language Processing*.
- Amir Sjarif, N. N., Mohd Azmi, N. F., Chuprat, S., Sarkan, H. M., Yahya, Y., & Sam, S. M. (2019). SMS Spam Message Detection using Term Frequency-Inverse Document Frequency and Random Forest Algorithm. *Procedia Computer Science*, 161, 509–515. <https://doi.org/10.1016/j.procs.2019.11.150>
- Arora, S., Li, Y., Liang, Y., Ma, T., & Risteski, A. (2016). A Latent Variable Model Approach to PMI-based Word Embeddings. *Transactions of the Association for Computational Linguistics*, 4, 385–399. https://doi.org/10.1162/tacl_a_00106
- Azlah, M. A. F., Chua, L. S., Rahmad, F. R., Abdullah, F. I., & Wan Alwi, S. R. (2019). Review on Techniques for Plant Leaf Classification and Recognition. *Computers*, 8(4), 77. <https://doi.org/10.3390/computers8040077>
- Bausch, J., Subramanian, S., & Piddock, S. (2021). A quantum search decoder for natural language processing. *Quantum Machine Intelligence*, 3(1), 16. <https://doi.org/10.1007/s42484-021-00041-1>
- Chatoui, H., & Ata, O. (2021). Automated Evaluation of the Virtual Assistant in Bleu and Rouge Scores. *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 1–6. <https://doi.org/10.1109/HORA52670.2021.9461351>
- Cheng, J., Dong, L., & Lapata, M. (2016). *Long Short-Term Memory-Networks for Machine Reading*.
- Cheng, W., Greaves, C., & Warren, M. (2006). From n-gram to skipgram to concgram. *International Journal of Corpus Linguistics*, 11(4), 411–433. <https://doi.org/10.1075/ijcl.11.4.04che>
- Choi, H., Kim, J., Joe, S., & Gwon, Y. (2021). Evaluation of BERT and ALBERT Sentence Embedding Performance on Downstream NLP Tasks. *2020 25th International Conference on Pattern Recognition (ICPR)*, 5482–5487. <https://doi.org/10.1109/ICPR48806.2021.9412102>
- Christian, H., Agus, M. P., & Suhartono, D. (2016). Single Document Automatic Text Summarization using Term Frequency-Inverse Document Frequency (TF-IDF). *ComTech: Computer, Mathematics and Engineering Applications*, 7(4), 285. <https://doi.org/10.21512/comtech.v7i4.3746>
- CHURCH, K. W. (2017). Word2Vec. *Natural Language Engineering*, 23(1), 155–162. <https://doi.org/10.1017/S1351324916000334>

- Datta, G., Joshi, N., & Gupta, K. (2021). *The BLEU Score for Automatic Evaluation of English to Bangla NMT* (pp. 399–407). https://doi.org/10.1007/978-981-33-4087-9_34
- de Jesus, O., & Hagan, M. T. (2002). Backpropagation through time for a general class of recurrent network. *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, 2638–2643. <https://doi.org/10.1109/IJCNN.2001.938786>
- Denis Rothman. (2021a). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*. Packt Publishing Ltd.
- Denis Rothman. (2021b). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*. Packt Publishing Ltd.
- Denis Rothman. (2021c). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*.
- Denis Rothman. (2021d). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*.
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*.
- Diego Lopez Yse. (n.d.). *Introduction to Transfer Learning*.
- Etqad Khan. (2020). *LSTM : What's the fuss about?* Analytics Vidhya.
- Ghatak, A. (2019). Recurrent Neural Networks (RNN) or Sequence Models. In *Deep Learning with R* (pp. 207–237). Springer Singapore. https://doi.org/10.1007/978-981-13-5850-0_8
- Graña Drummond, L. M., & Svaiter, B. F. (2005). A steepest descent method for vector optimization. *Journal of Computational and Applied Mathematics*, 175(2), 395–414. <https://doi.org/10.1016/j.cam.2004.06.018>
- Graves, A. (2012). Long Short-Term Memory. In *Supervised Sequence Labelling with Recurrent Neural Networks* (pp. 37–45). https://doi.org/10.1007/978-3-642-24797-2_4
- Havrlant, L., & Kreinovich, V. (2017). A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). *International Journal of General Systems*, 46(1), 27–36. <https://doi.org/10.1080/03081079.2017.1291635>

- HECHT-NIELSEN, R. (1992). Theory of the Backpropagation Neural Network**Based on “nonindent” by Robert Hecht-Nielsen, which appeared in Proceedings of the International Joint Conference on Neural Networks 1, 593–611, June 1989. © 1989 IEEE. In *Neural Networks for Perception* (pp. 65–93). Elsevier. <https://doi.org/10.1016/B978-0-12-741252-8.50010-8>
- JEREMY JORDAN. (n.d.). *Setting the learning rate of your neural network*. <https://www.jeremyjordan.me/nn-learning-rate/>.
- Jian Zheng, Cencen Xu, Ziang Zhang, & Xiaohua Li. (2017). Electric load forecasting in smart grids using Long-Short-Term-Memory based Recurrent Neural Network. *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, 1–6. <https://doi.org/10.1109/CISS.2017.7926112>
- Kai. (n.d.). *Find Similar Images Based On Locality Sensitive Hashing*. Westworld.
- Karunasingha, D. S. K. (2022). Root mean square error or mean absolute error? Use their ratio as well. *Information Sciences*, 585, 609–629. <https://doi.org/10.1016/j.ins.2021.11.036>
- Kenter, T., Borisov, A., & de Rijke, M. (2016). *Siamese CBOW: Optimizing Word Embeddings for Sentence Representations*.
- Li, B., & Han, L. (2013). *Distance Weighted Cosine Similarity Measure for Text Classification* (pp. 611–618). https://doi.org/10.1007/978-3-642-41278-3_74
- Li, Y., & Yang, T. (2018). *Word Embedding for Understanding Natural Language: A Survey* (pp. 83–104). https://doi.org/10.1007/978-3-319-53817-4_4
- Liu, B. (2020). Text sentiment analysis based on CBOW model and deep learning in big data environment. *Journal of Ambient Intelligence and Humanized Computing*, 11(2), 451–458. <https://doi.org/10.1007/s12652-018-1095-6>
- Llugin, R., Yacoubi, S. el, Fontaine, A., & Lupera, P. (2021). Comparison between Adam, AdaMax and Adam W optimizers to implement a Weather Forecast based on Neural Networks for the Andean city of Quito. *2021 IEEE Fifth Ecuador Technical Chapters Meeting (ETCM)*, 1–6. <https://doi.org/10.1109/ETCM53643.2021.9590681>
- Malik, P., & Baghel, A. S. (2016). An improvement in BLEU metric for English-Hindi machine translation evaluation. *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 331–336. <https://doi.org/10.1109/CCAA.2016.7813740>
- Mandar Deshpande. (2020). *The Transformer: A Quick Run Through*. Towards Data Science.

- Meister, C., Vieira, T., & Cotterell, R. (2020). Best-First Beam Search. *Transactions of the Association for Computational Linguistics*, 8, 795–809. https://doi.org/10.1162/tacl_a_00346
- Michael Phi. (2020). *Illustrated Guide to Transformers- Step by Step Explanation*.
- Mrinalini, K., P, V., & Thangavelu, N. (2022). SBSim: A Sentence-BERT Similarity-Based Evaluation Metric for Indian Language Neural Machine Translation Systems. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30, 1396–1406. <https://doi.org/10.1109/TASLP.2022.3161160>
- Murat H., S. (2006). A brief review of feed-forward neural networks. *Communications Faculty Of Science University of Ankara*, 50(1), 11–17. https://doi.org/10.1501/commua1-2_0000000026
- Nadkarni, P. M., Ohno-Machado, L., & Chapman, W. W. (2011). Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5), 544–551. <https://doi.org/10.1136/amiajnl-2011-000464>
- Neelakantan, A., Shankar, J., Passos, A., & McCallum, A. (2015). *Efficient Non-parametric Estimation of Multiple Embeddings per Word in Vector Space*.
- Nguyen, H. v., & Bai, L. (2011). *Cosine Similarity Metric Learning for Face Verification* (pp. 709–720). https://doi.org/10.1007/978-3-642-19309-5_55
- Qader, W. A., Ameen, M. M., & Ahmed, B. I. (2019). An Overview of Bag of Words;Importance, Implementation, Applications, and Challenges. *2019 International Engineering Conference (IEC)*, 200–204. <https://doi.org/10.1109/IEC47844.2019.8950616>
- Ramchoun, H., Amine, M., Idrissi, J., Ghanou, Y., & Ettaouil, M. (2016). Multilayer Perceptron: Architecture Optimization and Training. *International Journal of Interactive Multimedia and Artificial Intelligence*, 4(1), 26. <https://doi.org/10.9781/ijimai.2016.415>
- Robertson, S. (2004). Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation*, 60(5), 503–520. <https://doi.org/10.1108/00220410410560582>
- Senior, A., Heigold, G., Ranzato, M., & Yang, K. (2013). An empirical study of learning rates in deep neural networks for speech recognition. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 6724–6728. <https://doi.org/10.1109/ICASSP.2013.6638963>
- Sherstinsky, A. (2020). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404, 132306. <https://doi.org/10.1016/j.physd.2019.132306>

- Sutskever, I., Vinyals, O., & Le, Q. v. (2014). *Sequence to Sequence Learning with Neural Networks*.
- Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1), 43–62. [https://doi.org/10.1016/S0169-7439\(97\)00061-0](https://doi.org/10.1016/S0169-7439(97)00061-0)
- Tang, Y., Xu, J., Matsumoto, K., & Ono, C. (2016). Sequence-to-Sequence Model with Attention for Time Series Classification. *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, 503–510. <https://doi.org/10.1109/ICDMW.2016.0078>
- Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (n.d.). *Attention Is All You Need*.
- Vieira, S., Pinaya, W. H. L., & Mechelli, A. (2017). Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. *Neuroscience & Biobehavioral Reviews*, 74, 58–75. <https://doi.org/10.1016/j.neubiorev.2017.01.002>
- Vrbancic, G., & Podgorelec, V. (2020). Transfer Learning With Adaptive Fine-Tuning. *IEEE Access*, 8, 196197–196211. <https://doi.org/10.1109/ACCESS.2020.3034343>
- Wang, S.-C. (2003). Artificial Neural Network. In *Interdisciplinary Computing in Java Programming* (Vol. 743, pp. 81–100). Springer US. https://doi.org/10.1007/978-1-4615-0377-4_5
- Wolf, T., Sanh, V., Chaumond, J., & Delangue, C. (2019). *TransferTransfo: A Transfer Learning Approach for Neural Network Based Conversational Agents*.
- Xu, X., & Todorovic, S. (2016). Beam search for learning a deep Convolutional Neural Network of 3D shapes. *2016 23rd International Conference on Pattern Recognition (ICPR)*, 3506–3511. <https://doi.org/10.1109/ICPR.2016.7900177>
- Yang, L., Ma, S., Yang, H., & Tan, H. (2019). A Hierarchical Beam Search Algorithm with Better Performance for Millimeter-Wave Communication. *2019 2nd World Symposium on Communication Engineering (WSCE)*, 16–20. <https://doi.org/10.1109/WSCE49000.2019.9041150>
- Zhang, Z. (2018). Improved Adam Optimizer for Deep Neural Networks. *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 1–2. <https://doi.org/10.1109/IWQoS.2018.8624183>
- Zhao, R., & Mao, K. (2018). Fuzzy Bag-of-Words Model for Document Representation. *IEEE Transactions on Fuzzy Systems*, 26(2), 794–804. <https://doi.org/10.1109/TFUZZ.2017.2690222>

Appendix

9 Further Implementation Details

9.1 Hardware

Language processing requires high computational power to compute and develop a model. As we are using transformers to train our model, we can make the best of it by training them on a GPU. The GPU we have used to train our model is P6000 on AWS. With ample RAM, it can process a large batch of data to create a more generic model. It is recommended to have a minimum of 16 Gb of RAM, but we used around 24 Gb of ram, and the batch size that worked best was 32.

9.2 Languages and Frameworks

For the easy development of the model, we have used multiple Frameworks and APIs. To process and handle our data, we have used NumPy and Pandas libraries. To work with pre-trained models, we have used the Transformers library by Huggingface. As we are training our model using Pytorch, to make our training and validation easier, I have used the Pytorch Lightning library. This is a modified version of Pytorch. It inherits all the functions from the Pytorch.

9.3 Data Pre-Processing

9.3.1 Removing Docstring from The Code

Here the simplest way is to use regular expression and pass the code snippet through it.

```
def code_pre_proceeing(code):
    """
    input : unprocessed code
    output : clean code
    """

    # remove any html tag
    code = re.sub(r"<?.?span[^>]*>|<?.?code[^>]*>|<?.?p[^>]*>|<?.?hr[^>]*>|<?.?h[1-3][^>]*>|<?.?a[^>]*>|<?.?b[^>]*>|<?.?blockquote[^>]*>|<?.?del[^>]*>|<?.?dd[^>]*>|<?.?dl[^>]*>|<?.?dt[^>]*>|<?.?em[^>]*>|<?.?i[^>]*>|<?.?img[^>]*>|<?.?kbd[^>]*>|<?.?li[^>]*>|<?.?ol[^>]*>|<?.?pre[^>]*>|<?.?s[^>]*>|<?.?sup[^>]*>|<?.?sub[^>]*>|<?.?strong[^>]*>|<?.?strike[^>]*>|<?.?ul[^>]*>|<?.?br[^>]*>", "", code)

    # remove extra spaces and to lower cases
    code = ' '.join(code.split()).lower()

    return code
```

Figure 1: Remove HTML tag from code

9.3.2 Handling Outliers

To handle the outliers, we iterate every record in the row, tokenise it and see if the list of code tokens is less than the desired lengths.

```
# drop all the records that are greater than 256

long_code_index = [ i for i, length in enumerate( code_lengths ) if length > code_max_length ]

# drop all the records in the index
train.drop(index = long_code_index, inplace = True)

# reset index
train = train.reset_index().drop(labels = 'index', axis = 1)

# drop all the records that are greater than 64
cmt_lengths = [ len( tokenizer.encode( comment, add_special_tokens= False ) ) for comment in train.docstring.values ]

long_cmt_index = [ i for i, length in enumerate( cmt_lengths ) if length > cmt_max_length ]

# drop all the records in the index
train.drop(index = long_cmt_index, inplace = True)
```

Figure 2: Handle Outliers

9.4 Data Loader

For a clean code, I have written functions that recursively call each other to generate Input ids and Attention mask then convert them to Pytorch dataset and finally building a data loader.

```
# To tokenize the code
def code_tokenizer(data : list):
    """
    input : list of code to be tokenized
    output : input_ids and attention_mask tensor of type long
    """
    input_ids = []
    attention_mask = []

    for str in data:
        encoded = tokenizer.encode_plus( text= str , max_length= code_max_length, padding= 'max_length', truncation= True)
        input_ids.append(encoded['input_ids'])
        attention_mask.append(encoded['attention_mask'])

    # covert the input ids and the attention mask to type tensor
    input_ids = torch.LongTensor(input_ids)
    attention_mask = torch.LongTensor(attention_mask)

    return input_ids, attention_mask
```

Figure 3: Function to Generate Input ID's and Attention Mask

```

# creating a dataloader
def make_data_loader(data : pd.DataFrame ) -> torch.utils.data.DataLoader2 :
    """
    input : pandas data frame of code docstring pair
    output : DataLaoder that has src_input_ids, src_attention_mask, tgt_input_ids, tgt_attention_mask
    """

    assert 'code' in test.columns , "rename the function column name to code "
    assert 'docstring' in test.columns , "rename the description column name to docstring "

    SRC_input_ids, SRC_attention_mask = code_tokenizer(data.code.values)      # reurns input ids and attention mask of both
    source -> shape [N, S]
    TGT_input_ids, TGT_attention_mask = comment_tokenizer(data.docstring.values) # reurns input ids and attention mask of both
    source -> shape [N, T]

    data = torch.utils.data.TensorDataset( SRC_input_ids, SRC_attention_mask, TGT_input_ids, TGT_attention_mask ) # has four
    values in one variable

    sampler = torch.utils.data.RandomSampler(data)

    data_loader = torch.utils.data.DataLoader(dataset= data, batch_size= BATCH_SIZE, num_workers= 3, drop_last=True,
    sampler= sampler) # shape [N, S] -> [batchsize, sequence length]

    return data_loader

```

Figure 4: Generate Data Loader

9.5 Loading Pre-Trained Models

We load the pretrained weights and configuration to our encoder and decoder.

```

#pretrained configuration
config = transformers.RobertaConfig.from_pretrained("microsoft/codebert-base")

d_model = config.hidden_size
src_vocab_size = tokenizer.vocab_size
tgt_vocab_Size = tokenizer.vocab_size
pad_idx = tokenizer.pad_token_id
dropout = 0.5
src_max_length = code_max_length
tgt_max_length = cmt_max_length
epoch = 1000
lr = 5e-5

# Roberta Encoder layer
encoder = transformers.RobertaModel.from_pretrained(pretrained_model_name_or_path="microsoft/codebert-base", config = config )

# Decoder Transformers
decoder_layer = torch.nn.TransformerDecoderLayer(d_model=config.hidden_size, nhead=config.num_attention_heads)
decoder = torch.nn.TransformerDecoder(decoder_layer= decoder_layer, num_layers= 6)

# lightning model
model = Code_Comment_Generator(encoder, decoder, d_model, src_vocab_size, tgt_vocab_Size, pad_idx,
                                dropout, src_max_length, tgt_max_length, config, lr, epoch, tokenizer).to(device)

```

Figure 5: Loading Pretrained Weights

9.6 Function to generate comments for test cases

Here I have defined a function that takes in code as input and provides generated comments. To evaluate the generated comments using the BLEU score and cross encoder, we provide both actual and generated comments.

```
def generated_comment_report(code, actual_cmt = None):

    generated_comment = model.generate_comment(code_pre_proceeing(code))[0]

    print( f" code : \n { code } " )

    print( f" True comment : \n { actual_cmt }" )

    print( f" Generated Comment : \n { generated_comment }" )

    # calculate blue score
    blue_score = sentence_bleu([actual_cmt.lower()], generated_comment)
    print(f"the blue score for generated comment is {blue_score}")

    # calculate similarity
    similarity = np.average( check_similarity.predict([actual_cmt.lower(), generated_comment] ) )
    print(f"the similarity between sentences is {similarity}")
```

Figure 6: Function that takes code as a input and generates comments