Python Code Generation Using Deep Learning

Capstone Project Report

Amir Amirov

Nazarbayev University
Department of Electrical and Computer Engineering
School of Engineering and Digital Sciences

This project report was created on TexStudio editing platform using LaTeX. All the figures were drawn using draw.io online software tool.

NAZARBAYEV UNIVERSITY

Electrical and Computer Engineering
Nazarbayev University
http://www.nu.edu.kz

Title:
Python Code Generation using Deep Learning

Theme:
Sequence-to-sequence model

Project Period:
Spring 2024

Project Group:
ENG 400

Participant(s):
Amir Amirov

Supervisor(s):
Amin Zollanvari

Copies: 1

Page Numbers: 36

Date of Completion:
April 26, 2024

Abstract:

In this project, it is proposed to develop a sequence-to-sequence model for Python code generation using deep learning. The aim of this project is to investigate the feasibility of using deep learning to generate functional Python code automatically. It discusses the project's objectives, methodology, initial findings, and ethical considerations. This report references relevant literature. The significance of this project is in its attempt to propose a model with specific task with higher efficiency than general-purpose models.

Contents

Preface

This report delves into the innovative intersection of deep learning and software development, focusing on the automated generation of code. Chosen due to the increasing relevance and transformative potential of AI in programming, this work seeks to explore how deep learning can enhance and streamline the coding process.

Inspired by the challenges faced by developers in maintaining efficiency and accuracy amidst increasing software complexity, this project was conceived to harness the power of deep learning to automate and optimize the code generation process.

I would like to extend my deepest gratitude to Professor Amin Zollanvari for his invaluable guidance and supervision throughout the duration of this research project. His expertise and insights have been crucial in shaping the direction and execution of this work. Additionally, I am thankful for the access to the laboratory computers, which were essential in conducting the experiments and simulations presented in this report. Professor Zollanvari's support and mentorship have significantly enriched my learning experience and academic journey.

Nazarbayev University, April 26, 2024

Amir Amirov
<amir.amirov@nu.edu.kz>

Chapter 1

Introduction

In the burgeoning field of artificial intelligence, deep learning has emerged as a cornerstone for an array of complex computational tasks [1]. Among these, code generation represents a pivotal application, leveraging the capabilities of advanced neural network architectures to automate and optimize coding processes [2]. This report delves into the use of transformer-based models, specifically designed for code generation, highlighting their potential to enhance efficiency and accuracy in software development.

Transformers, introduced in the seminal paper "Attention is All You Need" by Vaswani et al., have revolutionized various domains of machine learning [3]. These models are characterized by their reliance on self-attention mechanisms, which weigh the significance of different parts of the input data, allowing for more nuanced understanding and generation of content [4]. Our project utilizes this architecture to specifically tailor a deep learning model for code generation. Unlike general models that perform a broad spectrum of tasks, our focused approach aims to refine the efficiency and output quality of generated code, serving as a testament to the adaptability and power of specialized transformer models.

The significance of this project lies in its attempt to bridge the gap between general-purpose models and specialized application needs. While general models offer broad capabilities, they often lack the fine-tuned precision required for specific tasks such as code generation. By developing a model dedicated to this task, we aim to achieve higher efficiency and effectiveness, setting a precedent for future specialized applications of transformer technology in the field of deep learning.

Chapter 2

Background

Initially, code generation tasks relied on recurrent neural networks (RNN) and Long Short-Term Memory networks (LSTM). These models were capable of handling sequences, making them suitable for generating code based on sequential data inputs [5]. However, they often struggled with long-range dependencies and computational efficiency [5]. "Long-range dependencies" refer to the requirement of the model to remember information from early in the sequence to use much later. For example, in language modeling, a pronoun appearing in a sentence might refer back to a noun mentioned several sentences earlier. Capturing these dependencies is crucial for understanding the context and maintaining the coherence of the generated text or code [6]. Handling long-range dependencies is challenging for standard RNNs due to the vanishing gradient problem, where gradients—used in training neural networks—become very small, effectively preventing the network from learning correlations between distant events in a sequence. The core operations in LSTMs are inherently sequential. Each step in processing a sequence depends on the completion of the previous step. This dependency limits the ability to parallelize operations, which is a key factor in speeding up neural network computations. Each LSTM unit includes multiple gates (input, forget, and output gates) and a memory cell. Each of these gates involves matrix multiplications and nonlinear activation functions. When processing each element of a sequence, these operations need to be computed, which increases the computational load compared to simpler architectures. LSTMs require substantial memory bandwidth and storage for parameters, previous states, and gradients during training. This can become a bottleneck in both training and inference phases, especially for long sequences or large models.Due to the above factors, training LSTMs can be resource-intensive and slow. They require significant computational resources, which can be costly and not efficient. The introduction of Transformer models marked a significant shift in code generation research. These models use self-attention mechanisms to weigh the influence of different words within the input data, regardless of their

positional distance [3]. As a result, Transformers handle long-range dependencies more effectively and offer greater parallelization, leading to faster training times and improved performance on large datasets.Recent studies have leveraged Transformer-based models for various code generation tasks. For example, the AlphaCode model by DeepMind uses an encoder-decoder Transformer architecture to generate code for competitive programming problems, achieving significant success rates and outperforming traditional code generation approaches [7]. Current research has expanded to include more nuanced approaches that combine deep learning models with other techniques to enhance code generation. For instance, some methods now incorporate retrieval-based approaches and post-processing steps to refine the generated code and align it more closely with human programming standards. These hybrid models have shown improved performance by leveraging both the predictive power of neural networks and contextual understanding from retrieval systems [8]. Despite these advancements, several challenges remain. These include improving the generalizability of models to handle diverse programming tasks, reducing the computational resources required, and enhancing the models' ability to understand and implement complex problem requirements. Future research is to focus on these areas, seeking to refine deep learning techniques and integrate them more seamlessly into practical programming environments [9].

Chapter 3

Data

In this project, three distinct datasets are used. The primary datasets employed are the Alpaca and XLCoST datasets, with the third being a merger of the two [10] [11]. As for training data, this merger combines the training, validation, and test sets of Alpaca with the training set of XLCoST, while exclusively adopting the validation and test sets from XLCoST for our merged dataset.

Dataset	Training	Validation	Test
Alpaca	10998	687	2062
XLCoST	9263	472	887
Overall	23010	472	887

Table 3.1: Composition of datasets used for training

Prompt	Python Code
Create a function to calculate the sum of a sequence of integers.	`def sum_sequence(sequence): sum = 0 for num in sequence: sum += return sum`
Write a function to generate random numbers between 0 and 9 that are divisible by 3	`def gen_divisible_number(): import random while True: process = random.randint(0,9) if process % 3 == 0: return process`
Develop a function in Python that prints out the Pascal's triangle for a given number of row.	`def pascal_trianlge(n): trow = [1] y = [0] for x in range(max(n,0)): print(trow) trow=[l+r for l,r in zip(trow + y, y + trow) ] return n>=1 pascal_triangle(5)`

Figure 1: Dataset examples.

Chapter 4

Methodology

This section will describe the detailed architecture of transformer. It is also important to note that the methodology encompasses more than just the transformer architecture; it also includes the preprocessing of data and the training process, among other critical steps, to ensure the model's effectiveness in generating code.

4.1 Transformer architecture

The transformer architecture is divided into two main parts: an encoder and a decoder (see Figure 1). The encoder's job is to read and understand the instructions or the prompt that describes what the user wants the code to do. It breaks down this information to make it easier for the machine to handle [12]. Encoder takes a prompt and converts it into context-rich representation. Next, the decoder uses this rich context, along with the sequence of the actual target output, to craft the code. It generates the code piece by piece, one token at a time, translating the user's intent into a functional script. This way, the encoder sets the stage, and the decoder brings the code to life. Nevertheless, before it some building blocks are required to be constructed.

4.2 Building the model

The following actions are required in order to construct the Transformer model [13].

1.Fundamental building blocks such as multi-head attention, position-wise feed-forward networks, tokenization, positional encoding are required to be defined.

2.Construction of the Encoder part of the model.

3.Construction of the Decoder part of the model.

4.Transformer network should be formed by connecting both parts of the model.

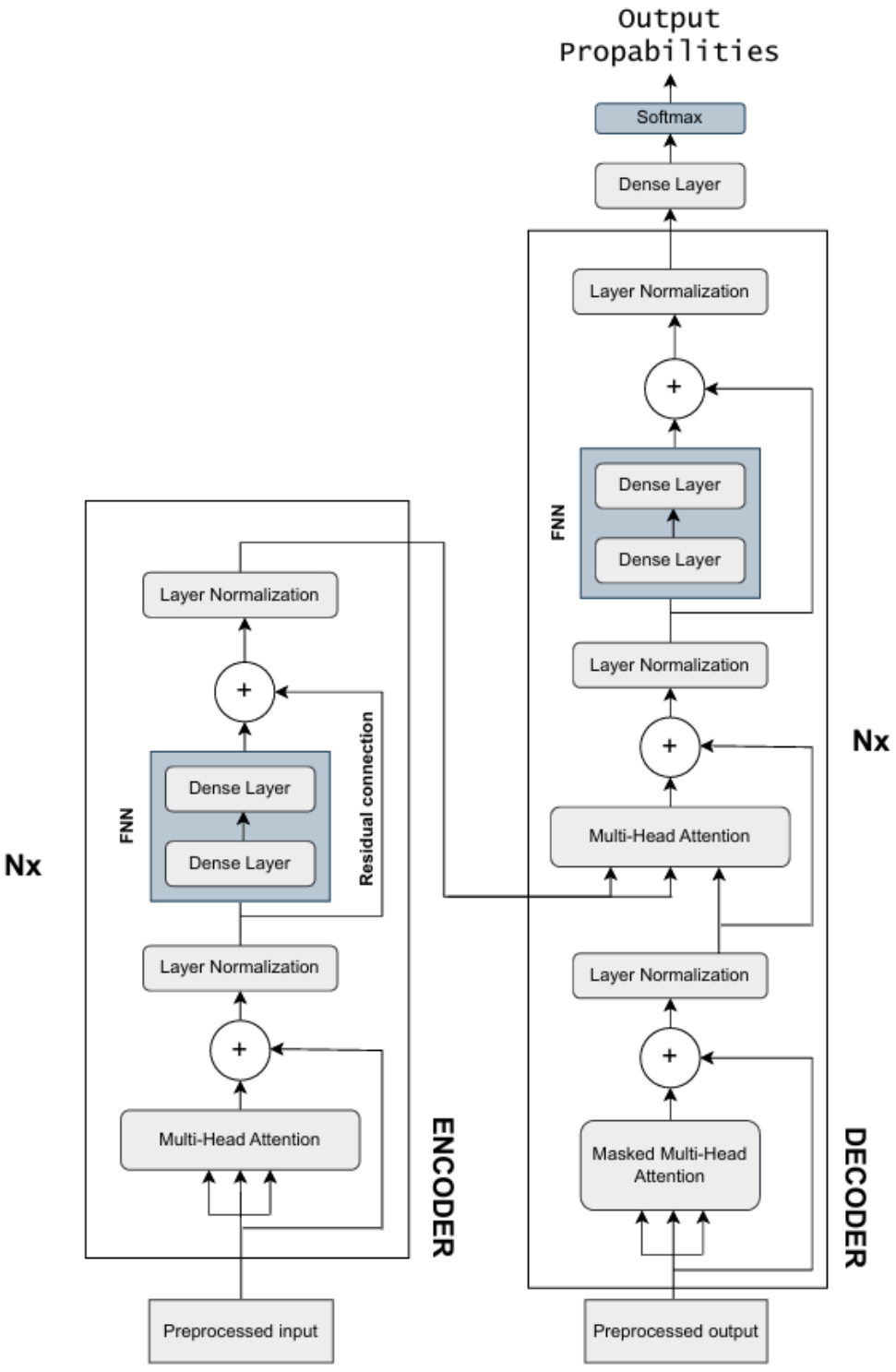

Figure 1: The architecture of the Transformer model

4.2.1 Preprocessing

In the sphere of deep learning, preprocessing is a critical step that transforms raw data into a clean, standardized format suitable for model training. This process is crucial because neural networks, such as the ones used for code generation, learn patterns from numerical representations of data [14]. Inconsistent or noisy input can significantly impede the learning process, leading to suboptimal performance. Therefore, the aim of preprocessing is to convert raw text into a uniform format, stripping away variations that do not contribute to the learning objective, such as capitalization, extraneous whitespace, or punctuation.

The preprocessing pipeline typically involves several steps: standardization, tokenization, and numericalization [15]. Standardization ensures that the text is consistent in case and formatting. Following this, tokenization breaks down the standardized text into atomic elements or tokens that the model can interpret. These tokens could be words, subwords, or characters, depending on the required task. Finally, numericalization maps each token to a unique index that represents it in the model's vocabulary.

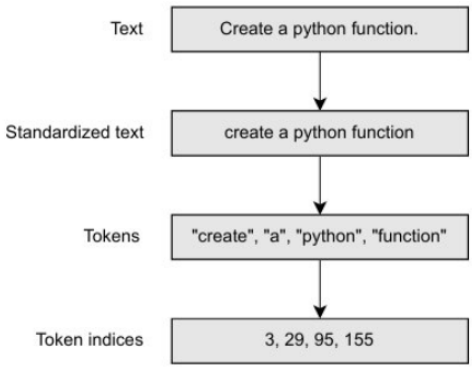

Figure 2: Diagram for self-attention.

The accompanying image exemplifies the preprocessing pipeline. It begins with the original text, "Create a python function.", which is standardized to lowercase, resulting in "create a python function". The text is then tokenized into individual words: "create", "a", "python", "function". Each word is subsequently converted into token indices, for instance, 3, 29, 95, 155, which the model uses to understand and process the text.

In our project, we leveraged the pretrained tokenizer from the CodeLlama-7b-Python model available on the Hugging Face platform [16]. This tokenizer is adept at parsing and understanding Python code, designed specifically for the

syntax and semantics of the Python language. It has been pretrained on a vast corpus of Python code, enabling it to recognize patterns and structures inherent to Python programming. The CodeLlama-7b-Python tokenizer simplifies the intricate process of understanding programming languages, a task that traditional natural language tokenizers are not equipped for. This specialization ensures that our model can effectively interpret code prompts and generate syntactically and semantically correct Python functions.

By employing such a tokenizer, we ensure that the input to our transformer model retains the crucial characteristics necessary for code generation while omitting superfluous details. This tailored preprocessing not only optimizes the training process but also enhances the model's ability to generate accurate and functional code.

4.2.2 Positional Encoding in Sequence-to-Sequence Models

In sequence-to-sequence models like transformers, understanding the order of words in a sentence is crucial for effective processing [3]. The technique employed to incorporate this positional information is known as "positional encoding."

Concept of Positional Encoding

The fundamental idea behind positional encoding is to enhance word embeddings with information about the position of each word in the sequence. This is achieved by augmenting the standard word vector with a position vector, representing the word's position in the sentence. The model is then expected to leverage this additional positional information to better understand the sequential relationships between words.

Simple Concatenation Approach

One straightforward approach is to concatenate the word's position directly to its embedding vector. However, this method has limitations as positions can be large integers, potentially disrupting the range of values in the embedding vector. Large input values are generally undesirable in neural networks.

Cosine-Based Positional Encoding

The "Attention is all you need" paper introduced a clever technique using cosine functions to encode word positions [3]. This method involves adding a vector with cyclically varying values in the range [-1, 1] to word embeddings.

Positional Embedding Implementation

```
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
def __init__(self, d_model, max_seq_length):
super(PositionalEncoding, self).__init__()

pe = torch.zeros(max_seq_length, d_model)
position = torch.arange(0, max_seq_length, dtype=torch.float)
 .unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(
 math.log(10000.0) / d_model))

pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)

self.register_buffer('pe', pe.unsqueeze(0))

def forward(self, x):
return x + self.pe[:, :x.size(1)]
```

Listing 4.1: PositionalEncoding Class

Mathematical Formulation

Let x be the input tensor of shape $(\text{batch\_size}, \text{seq\_length}, \text{d\_model})$. The positional encoding is calculated using sinusoidal functions [3]:

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

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

where $\text{PE}(pos, 2i)$ and $\text{PE}(pos, 2i+1)$ represent the positional encoding for even and odd indices, respectively.

Therefore, positional encoding is a technique crucial for incorporating word order information into sequence-to-sequence models. The chosen method, positional embedding, enriches word embeddings with learned position information. The implementation involves the addition of position embedding vectors to word embeddings, enhancing the model's ability to understand the sequential relationships within input sequences. The provided `PositionalEncoding` class effectively integrates this concept into the transformer architecture.

4.2.3 Attention Mechanism

Multi-head attention: When reading this report, a person may be skimming and devote more time to the more important part. The idea of a multi-attention mechanism is very similar to what people do. Not all information is equally useful for the task: the model should pay more attention to some features and less attention to other features [17] .

In the context of sequence-to-sequence models, the *multi-head attention* mechanism is an extension of the self-attention mechanism. Let's break down the key components and the mathematics involved.

Self-Attention

For the task of code generation, the application of attention mechanisms provides context for model. Self-attention, in particular, allows the model to weigh the importance of each token within a sequence, enhancing its understanding of the syntactic and semantic structure of the code.

Self-attention is like a system that helps a word in a sentence pay attention to other important words to better understand what it means in that specific context. Take the sentence "Bus arrived at the station on time." If we focus on the word "station," we might wonder what type of station it is. Is it for trains, radios, or maybe even space, like the International Space Station? Self-attention helps figure this out by looking at the word "station" and considering the other words around it to get the right meaning.

The diagram is a visualization of how the self-attention mechanism in neural networks processes the sentence "Bus arrived at the station on time." The sentence is broken down into its individual words: 'bus,' 'arrived,' 'at,' 'the,' 'station,' 'on,' 'time.' Each word is represented as a token vector. These vectors are typically generated through an embedding process that turns words into numerical representations that capture some of their meaning. The diagram shows a matrix where each cell represents the attention score between two words. For instance, there's a high attention score (0.8) between 'bus' and 'station,' suggesting that the model pays more attention to how these two words are related in the context of the sentence. This is logical since 'bus' and 'station' have a direct connection. The attention scores are then scaled and normalized through a softmax function so that they add up to 1. This makes it easier for the model to weigh the importance of each word when considering 'station.' Each token vector is multiplied by the attention scores to emphasize some words and de-emphasize others. This step creates weighted token vectors, which are then used to create a context-aware representation of each word. The weighted vectors for 'station' are summed to create a single context-aware vector that represents 'station' in the context of the whole sentence. This final vector for 'station' captures not just the meaning of the word by itself,

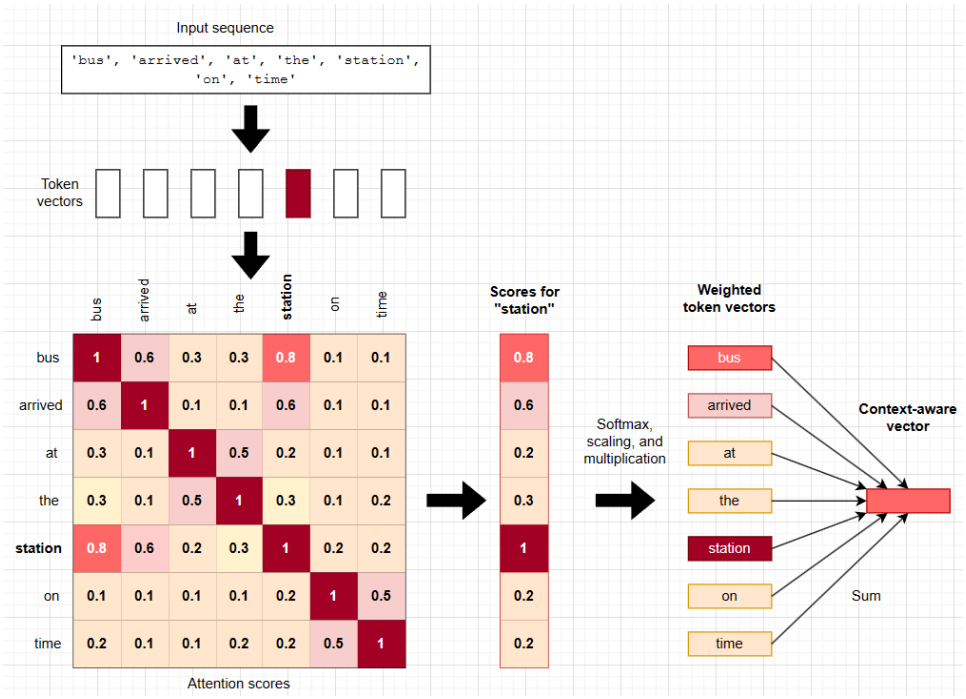

Figure 3: Diagram for self-attention.

but its meaning within the sentence, reflecting the influence of related words like 'bus' and 'arrived.'

The fundamental operation in this context is the scaled dot-product self-attention, expressed as $SA^{d}_{V,K,Q}(\tilde{X}_i)$, which maps an input sequence into an output space with refined representations [18]:

$$SA^{d}_{V,K,Q}(\tilde{X}_i) = V\tilde{X}_i \cdot \text{softmax}\left(\frac{\tilde{X}_i^T K^T Q \tilde{X}_i}{\sqrt{d_k}}\right) \tag{4.1}$$

Here, V, K, and Q are the learnable projection matrices specific to the domain of code synthesis. d_k represents the dimension of the key/query vectors, which is tuned during the training process to encapsulate the complexity of the code tokens. The output is a set of context vectors, which capture the intricate dependencies within the code.

Multi-Head Attention

Extending the concept of self-attention, multi-head self-attention allows the model to concurrently process the information through different representation subspaces at different positions [3]. Now, in multi-head attention, this process is performed

multiple times in parallel, each with a separate set of learnable projections for Q, K, and V. Each set of projections is often referred to as a "head." Let's denote the number of heads as h.

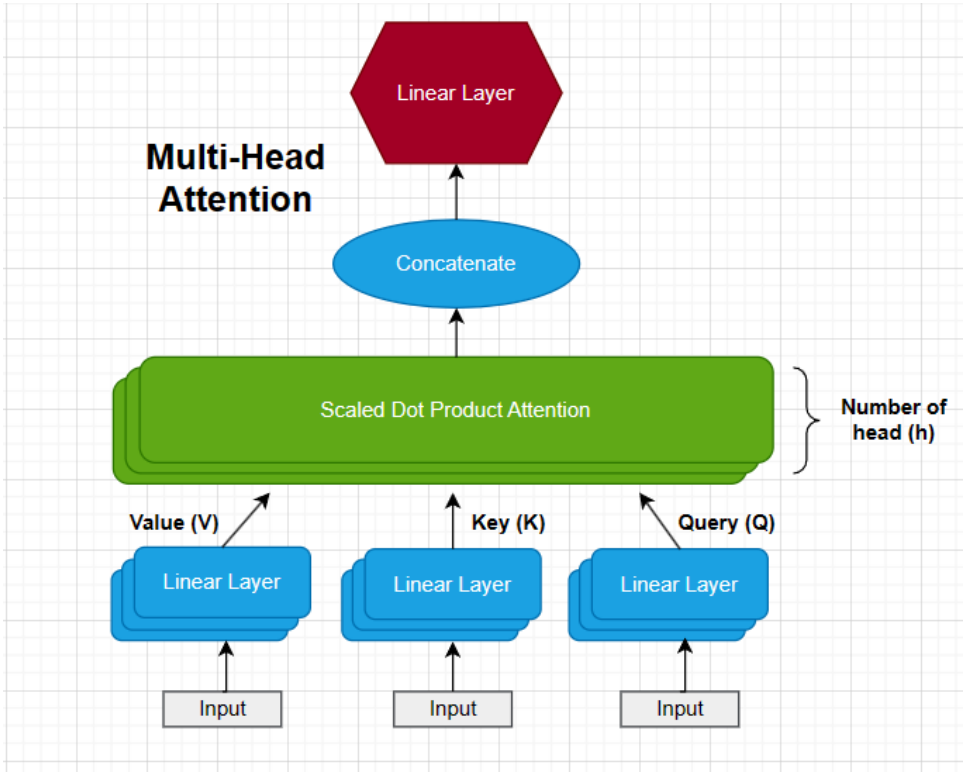

Figure 4: Diagram for Multi-Head Attention mechanism.

The diagram you see is a depiction of a Multi-Head Attention mechanism, an essential part of Transformer neural network models. It starts with the input data, which, in our case, is numerical version of prompt. This data goes through several linear layers that transform it into three distinct formats: Queries, Keys, and Values. These transformations are crucial because they prepare the input for the attention process, allowing the model to handle different parts of the data in separate ways.

The heart of the system is the Scaled Dot Product Attention. It computes attention scores by comparing Queries and Keys, determining how much focus should be placed on each part of the input for every Value. Scaling is a technique used here to prevent issues during training that could slow down learning.

A unique feature of this mechanism is the multiple 'heads' in the Multi-Head Attention block. Each head attends to different parts of the input, enabling the model to consider various pieces of information simultaneously. It's like having several different models looking at the same problem from different angles, all at

once.

After each head has analyzed the data, their outputs are merged through concatenation. This step is essential because it brings together the diverse perspectives captured by each head. Finally, another linear layer is used for compatibility purposes. This composite output is then ready for further processing within the model.

This parallel processing is mathematically denoted as [18]:

$$MHSA^{d_h}(\tilde{X}_i) = W \left[SA^{d}_{V_1,K_1,Q_1}(\tilde{X}_i)^T, \ldots, SA^{d}_{V_h,K_h,Q_h}(\tilde{X}_i^T) \right]^T \tag{4.2}$$

W is another learnable matrix that integrates the information from multiple self-attention mechanisms, each one providing a unique perspective on the input code sequence. h denotes the number of distinct attention mechanisms, or heads, and d_h specifies the dimensionality of the output feature space of the multi-head self-attention operation. Through these mechanisms, the model can generate a more nuanced and comprehensive representation of code, which is essential for accurately generating new code segments.

Projection

For each head i ($i = 1, 2, \ldots, H$), the input X is projected into three spaces: Q_i, K_i, and V_i. These projections are achieved by learned linear transformations (linear dense layers shown in Figure 4):

$$\begin{aligned} Q_i &= XW_{Qi}, \\ K_i &= XW_{Ki}, \\ V_i &= XW_{Vi}, \end{aligned}$$

where W_{Qi}, W_{Ki}, and W_{Vi} are the weight matrices for the i-th head.

Interpretation

- **Learnability:** The presence of learnable dense projections (W_{Qi}, W_{Ki}, W_{Vi}) enables the model to capture complex patterns and dependencies in the data or find those attention score shown earlier (see Figure 3).
- **Independence of Heads:** Each head specializes in different aspects of the input, allowing the model to attend to different patterns simultaneously. This can be especially beneficial for capturing diverse features in the data.

Multi-Head Attention Implementation in PyTorch

Here is the implementation of the `MultiHeadAttention` class in PyTorch:

```
1
2 class MultiHeadAttention(nn.Module):
3 def __init__(self, d_model, num_heads):
4 super(MultiHeadAttention, self).__init__()
5 # Ensure that the model dimension (d_model) is divisible by the
      number of heads
6 assert d_model % num_heads == 0, "d_model must be divisible by
     num_heads"
7
8 # Initialize dimensions
9 self.d_model = d_model # Model's dimension
10 self.num_heads = num_heads # Number of attention heads
11 self.d_k = d_model // num_heads # Dimension of each head's key,
      query, and value
12
13 # Linear layers for transforming inputs
14 self.W_q = nn.Linear(d_model, d_model) # Query transformation
15 self.W_k = nn.Linear(d_model, d_model) # Key transformation
16 self.W_v = nn.Linear(d_model, d_model) # Value transformation
17 self.W_o = nn.Linear(d_model, d_model) # Output transformation
18
19 def scaled_dot_product_attention(self, Q, K, V, mask=None):
20 # Calculate attention scores
21 attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(
     self.d_k)
22
23 # Apply mask if provided (useful for preventing attention to
     certain parts like padding)
24 if mask is not None:
25 attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
26
27 # Softmax is applied to obtain attention probabilities
28 attn_probs = torch.softmax(attn_scores, dim=-1)
29
30 # Multiply by values to obtain the final output
31 output = torch.matmul(attn_probs, V)
32 return output
33
34 def split_heads(self, x):
35 # Reshape the input to have num_heads for multi-head attention
36 batch_size, seq_length, d_model = x.size()
37 return x.view(batch_size, seq_length, self.num_heads, self.d_k)
     .transpose(1, 2)
38
39 def combine_heads(self, x):
40 # Combine the multiple heads back to original shape
41 batch_size, _, seq_length, d_k = x.size()
```

```
return x.transpose(1, 2).contiguous().view(batch_size,
    seq_length, self.d_model)

def forward(self, Q, K, V, mask=None):
# Apply linear transformations and split heads
Q = self.split_heads(self.W_q(Q))
K = self.split_heads(self.W_k(K))
V = self.split_heads(self.W_v(V))

# Perform scaled dot-product attention
attn_output = self.scaled_dot_product_attention(Q, K, V, mask)

# Combine heads and apply output transformation
output = self.W_o(self.combine_heads(attn_output))
return output
```

Listing 4.2: MultiHeadAttention Class

The multi-head attention mechanism enhances the expressive power of the model, enabling it to learn intricate relationships within the sequences it processes.

4.2.4 Residual connection and normalization

The multi-head attention mechanism transforms these embeddings in a way that incorporates contextual information from other tokens in the sequence. The transformed embeddings (the output of the attention mechanism) are then added to the original embeddings. This step is the residual connection or identity skip-connection. Residual connection is generally good practice but with good normalization. The addition allows the gradients to flow more easily through the network during training (helping to prevent the vanishing gradient problem), and normalization helps to stabilize the learning process.

Identity skip-connection mathematically represented as [18]:

$$\mathrm{SKP}(LAY(Y)) = Y + LAY(Y)$$

where $LAY(Y)$ denotes the layer operation on the input matrix $Y \in \mathbb{R}^{a\times b}$, and $\mathrm{SKP}(\mathrm{LAY}(Y)) : \mathbb{R}^{a\times b} \to \mathbb{R}^{a\times b}$ denotes the skip connection operation.

Layer Normalization (LN) comes into play subsequent to the residual connection.

After a multi-head self-attention operation (MHSA), the Transformer applies these principles as follows [18]:

$$X_i = LN(\mathrm{SKP}(MHSA(\hat{X}_i)));$$

This equation demonstrates how the output of the MHSA is first regulated by the skip connection and then normalized. The LN thus acts as a form of feature

scaling, standardizing the input's feature space to facilitate smoother and more stable gradient descent during model training.

These techniques are not only intrinsic to maintaining training stability but also serve to enhance the model's ability to generalize from the data it is trained on, thus improving overall performance.

4.2.5 Position-Wise Feed-Forward Networks

After the application of the multi-head attention mechanism, the next crucial component in the sequence-to-sequence model is the Position-Wise Feed-Forward Networks. These networks play a crucial role in further processing the information obtained through attention mechanisms, contributing to the overall effectiveness of the transformer architecture.

In Transformer model architectures, the encoder and decoder encompasses a distinct substructure called the position-wise fully connected feed-forward network. This design is imperative for individually and concurrently processing each sequence position. Represented by a matrix $Y \in \mathbb{R}^{a \times b}$, this structure embodies the sequence information.

Given input Y, the position-wise feed-forward network applies a transformation expressed as $FFN^{(s)}(Y)$, functioning independently on each vector y_k (with k spanning from 1 to b, representing the columns of Y) [18]:

$$FFN^{(s)}(Y) = [g(y_1), \ldots, g(y_b)],$$

where each $g(y_k)$ is defined by [18]:

$$g(y_k) = W_2\sigma(W_1 y_k + b_1) + b_2,$$

In this context, $\sigma(\cdot)$ represents an activation function such as ReLU, and $W_1 \in \mathbb{R}^{r \times a}, W_2 \in \mathbb{R}^{s \times r}, b_1 \in \mathbb{R}^{r \times 1}$, and $b_2 \in \mathbb{R}^{s \times 1}$ are the trainable parameters of the network. Here, r is a tuning hyperparameter indicating the internal layer dimensionality.

The superscript s in $FFN^{(s)}(Y)$ emphasizes the output vector size, aligned with the dimensionality. In the encoder segment of the Transformer, the feed-forward network yields an output O_i for each respective input X_i, formalized as [18]:

$$O_i = LN(SKP(FFN^{(c)}(X_i))).$$

This formulation integrates skip-connections (denoted by SKP) as a core element, as described in equation (17). These connections play a crucial role in the architecture, promoting information retention across layers and mitigating the vanishing gradient issue by allowing direct input progression to subsequent layers.

The utilization of layer normalization (LN) concurrently normalizes the activations, aiding in maintaining consistent training behavior.

Crucial to the Transformer's efficacy, these feed-forward networks facilitate the parallel treatment of sequence data. When amalgamated with the attention mechanism, they furnish a powerful framework suitable for complex tasks like deep learning-based code generation. The process culminates in a classification layer, equipped with a softmax activation, that distributes probabilities across different potential classes.

PositionWiseFeedForward Class

```
import torch.nn as nn

class PositionWiseFeedForward(nn.Module):
def __init__(self, d_model, d_ff):
super(PositionWiseFeedForward, self).__init__()
self.fc1 = nn.Linear(d_model, d_ff)
self.fc2 = nn.Linear(d_ff, d_model)
self.relu = nn.ReLU()

def forward(self, x):
return self.fc2(self.relu(self.fc1(x)))
```

Listing 4.3: PositionWiseFeedForward Class

Thus, the `PositionWiseFeedForward` class encapsulates a position-wise feed-forward neural network, incorporating two linear layers with a rectified linear unit (ReLU) activation function in between. This network is applied independently to each position in the input sequence within the context of transformer models. It serves as a crucial component for post-processing the information obtained through attention mechanisms, enhancing the model's ability to capture intricate patterns in the data.

4.2.6 Construction of Encoder and Decoder

With our foundational components in place, including Multi-Head Attention, Position-wise Feed-Forward Networks, and Positional Encoding, we can now proceed to construct the encoder for our sequence-to-sequence model (see Figure 5). The encoder is a critical element in the transformer architecture, responsible for processing input sequences and capturing meaningful representations. Let's delve into the construction details.

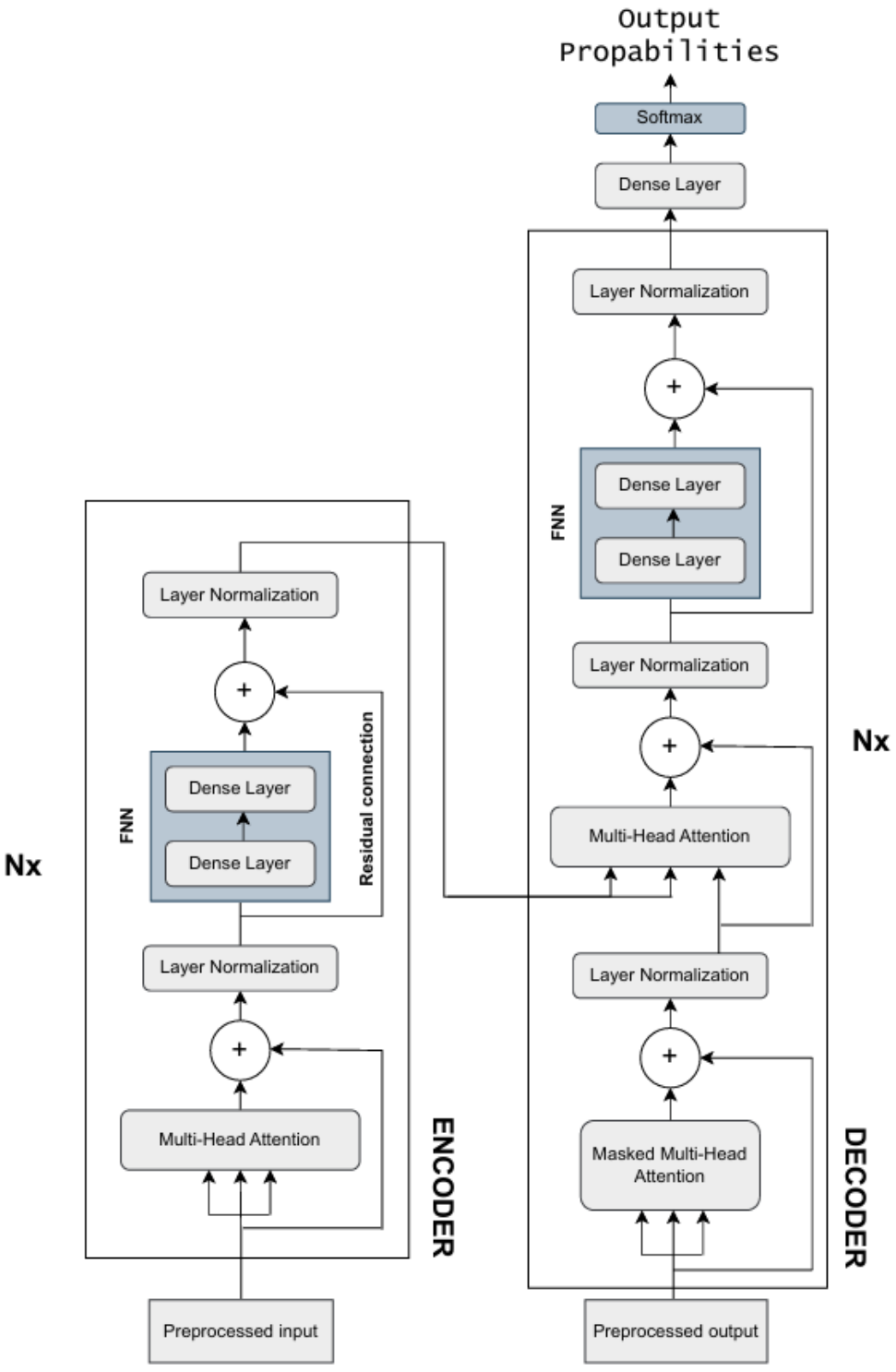

Figure 5: The architecture of the Transformer model

Encoder Architecture Overview

The encoder in our transformer architecture serves as the initial processing unit for code generation. Designed with a series of six stacked layers, meaning that we have 6 encoders. Each possessing a feature size of 128, it is finely attuned to comprehend and contextualize the preprocessed input. At the heart of each layer lies a multi-head attention mechanism, with 8 distinct attention heads working together to thoroughly examine and identify interdependencies in the input data. This intricate system allows the encoder to craft a multi-dimensional representation of the code prompt, which encapsulates both the explicit instructions and implicit coding patterns.

Following the attention process, each layer employs a position-wise feed-forward network with a dimensionality of 1024. This network further processes the information, enabling the model to grasp complex coding constructs. Residual connections around these networks, along with subsequent layer normalization, preserve and enhance information flow, ensuring that even subtle syntactic nuances are retained throughout the processing stages. Additionally, dropout is utilized to prevent overfitting by randomly dropping some connections during training. The encoder ensures a rich and consistent representation, setting a strong foundation for the decoder to generate precise and efficient code.

Implementation of EncoderLayer

```
import torch.nn as nn

class EncoderLayer(nn.Module):
def __init__(self, d_model, num_heads, d_ff, dropout):
super(EncoderLayer, self).__init__()
self.self_attn = MultiHeadAttention(d_model, num_heads)
self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
self.norm1 = nn.LayerNorm(d_model)
self.norm2 = nn.LayerNorm(d_model)
self.dropout = nn.Dropout(dropout)

def forward(self, x, mask):
attn_output = self.self_attn(x, x, x, mask)
x = self.norm1(x + self.dropout(attn_output))
ff_output = self.feed_forward(x)
x = self.norm2(x + self.dropout(ff_output))
return x
```

Listing 4.4: EncoderLayer Class

Flow

1. **Self-Attention:** attn_output $=$ self_attn(x, x, x, mask)
2. **Addition and Normalization:** $x = \text{norm1}(x + \text{dropout}(\text{attn\_output}))$
3. **Position-wise Feed-Forward:** ff_output $=$ feed_forward(x)
4. **Addition and Normalization:** $x = \text{norm2}(x + \text{dropout}(\text{ff\_output}))$

In summary, the construction of the encoder involves stacking multiple layers, each containing a self-attention mechanism and a position-wise feed-forward network. These components work in tandem to capture hierarchical and position-aware representations of the input sequence. The provided `EncoderLayer` class encapsulates this architecture, allowing for flexible and effective encoding of sequential data in our transformer model.

Construction of Decoder

With the foundational components of Multi-Head Attention, Position-wise Feed-Forward Networks, and Positional Encoding, we are equipped to construct the decoder for our sequence-to-sequence model. The decoder plays a pivotal role in generating target sequences by attending to the encoder's output. Let's explore the construction details based on your code and transformer architecture principles.

Decoder Architecture Overview

The decoder in our model is composed of 6 layers, meaning there are 6 decoder models. With each layer designed to handle a feature size of 256 for the inputs. Like the encoder, the decoder is equipped with 8 attention heads, ensuring a meticulous cross-referencing of the context provided by the encoder's output. This layered structure is key for the model to incrementally build up the final output, which is the code itself. The decoder in a transformer takes the context-rich information from the encoder and begins the task of generating the output, one token at a time. It learns to predict the next element in the sequence by analyzing the target's previous elements and the contextual information provided by the encoder. This approach is known as teacher forcing.

Teaching forcing is a training strategy for sequence generation models, including our decoder in the transformer architecture. This technique involves using the actual output from the previous time step as an input to the model during the next step, rather than using the model's own prediction. The rationale behind teacher forcing is to accelerate and stabilize training by guiding the model with the correct sequence, especially in the early stages when the model's predictions can be largely inaccurate. To prevent the model from 'peeking' ahead, which could lead

to cheating, we employ an additional attention mechanism within the decoder. This mechanism is specifically designed to mask future tokens in the sequence. By doing so, the model is constrained to use only the information of the preceding tokens and the current context to predict the next token, thus emulating the actual conditions under which it will generate code during inference. The model configuration reflects these design choices, with a feature size of 128 for encoder inputs and 256 for decoder inputs, ensuring that the representations are sufficiently detailed for complex code generation tasks. The inclusion of eight attention heads allows the decoder to pay detailed attention to different parts of the input sequence simultaneously. Moreover, each layer contains a feed-forward network with a dimensionality of 1024, providing the necessary computational space to process and generate intricate coding patterns.

Looking at the example provided, let's see how teaching forcing works. You provide the model with an input sequence. In the Figure 6, the input sequence is the text "Create a function to add two numbers." The encoder processes the entire input sequence and creates a context vector (or a set of vectors), which is a representation of the input. The training begins with a start token (usually denoted as [start] or something similar). This signals the decoder to start generating the output. Instead of using the decoder's previous output as the next input (which would be the case in inference), forced teaching uses the correct next token from the training dataset. In your image, even though the decoder just outputted "def", the next input token is the correct next piece of the code `add_numbers(a, b):` which is provided externally. This process continues one token at a time, with the decoder being 'forced' to predict the next token with the correct previous token always being provided. This way, the model is trained to predict the next token in the sequence given the correct history of tokens.

During the inference phase, the process is similar, but the decoder no longer has access to the target sequence. It must generate the code solely based on the encoder's output and what it has produced sequentially. Starting with a token indicating the start of generation, it continues to predict subsequent tokens until it reaches a point where it predicts an end token, signaling the completion of code generation. This sequential and masked approach enables the model to generate code independently, reflecting the learned patterns from the training phase without any external guidance.

Implementation of DecoderLayer

```
import torch.nn as nn

class DecoderLayer(nn.Module):
def __init__(self, d_model, num_heads, d_ff, dropout):
super(DecoderLayer, self).__init__()
```

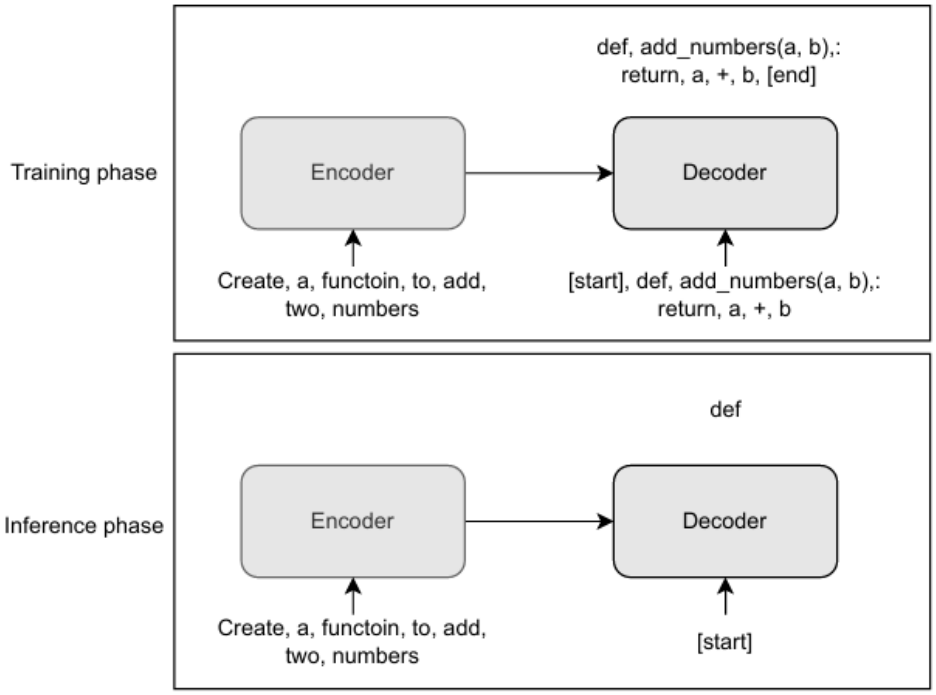

Figure 6: The decoder part of the Transformer model

```
6   self.self_attn = MultiHeadAttention(d_model, num_heads)
7   self.cross_attn = MultiHeadAttention(d_model, num_heads)
8   self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
9   self.norm1 = nn.LayerNorm(d_model)
10  self.norm2 = nn.LayerNorm(d_model)
11  self.norm3 = nn.LayerNorm(d_model)
12  self.dropout = nn.Dropout(dropout)
13
14  def forward(self, x, enc_output, src_mask, tgt_mask):
15  attn_output = self.self_attn(x, x, x, tgt_mask)
16  x = self.norm1(x + self.dropout(attn_output))
17  attn_output = self.cross_attn(x, enc_output, enc_output,
     src_mask)
18  x = self.norm2(x + self.dropout(attn_output))
19  ff_output = self.feed_forward(x)
20  x = self.norm3(x + self.dropout(ff_output))
21  return x
```

Listing 4.5: DecoderLayer Class

Flow

1. **Masked Self-Attention:** $\text{attn\_output} = \text{self\_attn}(x, x, x, \text{tgt\_mask})$
2. **Addition and Normalization:** $x = \text{norm1}(x + \text{dropout}(\text{attn\_output}))$
3. **Cross-Attention:** $\text{attn\_output} = \text{cross\_attn}(x, \text{enc\_output}, \text{enc\_output}, \text{src\_mask})$
4. **Addition and Normalization:** $x = \text{norm2}(x + \text{dropout}(\text{attn\_output}))$
5. **Position-wise Feed-Forward:** $\text{ff\_output} = \text{feed\_forward}(x)$
6. **Addition and Normalization:** $x = \text{norm3}(x + \text{dropout}(\text{ff\_output}))$

Hence, the decoder is constructed by stacking multiple layers, each containing masked self-attention, cross-attention, and position-wise feed-forward mechanisms. These components collectively allow the decoder to generate target sequences based on the learned representations from the encoder. The provided `DecoderLayer` class encapsulates this architecture, facilitating the effective decoding of sequences in our transformer model.

Transformer Network: Connecting the Encoder and Decoder Parts

The Transformer network provides the seamless connection between the Encoder and Decoder, defining a forward pass that transforms source and target sequences into meaningful output predictions. This section details the essential steps of this process.

Input Embedding and Positional Encoding

It begins with the embedding of source and target sequences using dedicated embedding layers. These embedded sequences are then enriched with positional encodings, providing vital information about the position and order of tokens within the sequences.

Encoder Layers

The source sequence undergoes a transformative process as it traverses through the Encoder Layers. These layers collectively process the source sequence, with the final output representing a distilled and refined representation of the input.

Decoder Layers

Simultaneously, the target sequence, along with the encoder's output, navigates through the Decoder Layers. This collaborative journey results in the generation of the decoder's output, capturing the intricate dependencies within the target sequence.

Final Layer

The decoder's output undergoes further refinement through a Final Linear Layer, a fully connected layer that maps the output to the target vocabulary size. This step is needed for compatibility purposes.

Output

The final output is encapsulated in a tensor, representing the model's predictions for the target sequence. This tensor carries the distilled knowledge and insights gained from the input sequences.

Code Implementation

```
class Transformer(nn.Module):
def __init__(self, src_vocab_size, tgt_vocab_size, d_model,
    num_heads, num_layers, d_ff, max_seq_length, dropout):
super(Transformer, self).__init__()
self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
self.positional_encoding = PositionalEncoding(d_model,
    max_seq_length)

self.encoder_layers = nn.ModuleList([EncoderLayer(d_model,
    num_heads, d_ff, dropout) for _ in range(num_layers)])
self.decoder_layers = nn.ModuleList([DecoderLayer(d_model,
    num_heads, d_ff, dropout) for _ in range(num_layers)])

self.fc = nn.Linear(d_model, tgt_vocab_size)
self.dropout = nn.Dropout(dropout)

def generate_mask(self, src, tgt):
src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)
seq_length = tgt.size(1)
nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length,
    seq_length), diagonal=1)).bool()
tgt_mask = tgt_mask & nopeak_mask
```

```
return src_mask, tgt_mask

def forward(self, src, tgt):
src_mask, tgt_mask = self.generate_mask(src, tgt)
src_embedded = self.dropout(self.positional_encoding(self.
    encoder_embedding(src)))
tgt_embedded = self.dropout(self.positional_encoding(self.
    decoder_embedding(tgt)))

enc_output = src_embedded
for enc_layer in self.encoder_layers:
enc_output = enc_layer(enc_output, src_mask)

dec_output = tgt_embedded
for dec_layer in self.decoder_layers:
dec_output = dec_layer(dec_output, enc_output, src_mask,
    tgt_mask)

output = self.fc(dec_output)
return output
```

Listing 4.6: Transformer Class Implementation

4.2.7 Training the Transformer Model

Training the Transformer model involves choices for the loss function, optimizer, and evaluation metric. Each plays a crucial role in shaping the model's performance. Let's delve into the specifics of the loss function, optimizer, and the metrics.

Loss Function: Cross-Entropy Loss

The Cross-Entropy Loss, also known as log loss, is a widely employed loss function for classification tasks, including sequence-to-sequence models[19]. Mathematically, it measures the dissimilarity between the predicted probability distribution and the true distribution of the target sequence. In the context of the Transformer model, the Cross-Entropy Loss is particularly adept at quantifying the disparity between predicted and actual token distributions.

The formula for the Cross-Entropy Loss is given by [20]:

$$\text{Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(p_{ij})$$

Here,

N : number of training instances,
C : number of classes (vocabulary size),
y_{ij} : indicator function,
p_{ij} : predicted probability.

Optimizer: Adam

The Adam optimizer, an adaptive learning rate optimization algorithm, combines the strengths of both the AdaGrad and RMSProp algorithms [21]. Its adaptive nature enables it to dynamically adjust the learning rate for each parameter during training.

Mathematically, the update rule for the Adam optimizer is given by [21]:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$
$$\theta_t = \theta_{t-1} - \frac{\alpha \cdot m_t}{\sqrt{v_t} + \epsilon}$$

Here, α is the learning rate, β_1 and β_2 are smoothing parameters, m_t and v_t are moments of the gradients, g_t is the gradient, θ_t is the parameter being updated, and ϵ is a small constant to prevent division by zero.

Evaluation Metrics for Model Performance

Evaluating the performance of machine learning models in natural language processing (NLP) and code generation tasks requires precise and appropriate metrics. These metrics enable a quantifiable comparison of machine-generated outputs against human or ideal references. In this project, we employ BLEU scores and CodeBERTScore as our primary evaluation tools. These metrics are selected for their relevance to the respective fields of text and code generation and their ability to provide insights into the quality of the generated outputs.

BLEU Scores

Description:
BLEU (Bilingual Evaluation Understudy) is a metric originally designed to evaluate the quality of text translated by machine to another language against one or more human-produced reference translations [22]. It is also extensively used in other text generation tasks. BLEU measures the correspondence between a machine's output and that of a human at the level of word n-grams, providing a score from 0 to 1, where 1 is a perfect match with the reference.

Mathematical Formula:
The BLEU score for a machine translation is calculated as follows [22]:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Where:

- p_n is the precision of n-grams, calculated as the ratio of the number of n-gram matches between the machine output and reference to the total number of n-grams in the machine output.
- w_n are weights assigned to each n-gram size (usually uniform).
- BP (brevity penalty) penalizes short machine outputs and is defined as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

  Here, c is the length of the candidate translation and r is the effective reference corpus length.

Justification for Choice:
BLEU scores are chosen due to their widespread acceptance and usage in the evaluation of text generation tasks, including translation and content creation. Their quantitative nature allows for straightforward comparison across different models and benchmarks.

Example of How It Works:
For instance, consider a machine-generated sentence "The black cat sat on the mat." and a reference sentence "A black cat was sitting on the mat." The BLEU score would quantify the overlap in n-grams (e.g., bigrams like "black cat", "on the") between these two sentences to provide a measure of quality.

CodeBERTScore

Description:
CodeBERTScore is an innovative metric designed for assessing code generation tasks. It builds upon the concept of BERTScore, incorporating the encoding of both the generated code and the natural language input, thus ensuring the consistency between the generated code and its context. The evaluation performed across multiple programming languages demonstrates CodeBERTScore's superior

correlation with human preferences and functional correctness of the generated code [23].

Mathematical Formulation:
Precision P and Recall R for CodeBERTScore are derived as follows [23]:

$$P = \frac{1}{|y_{\text{gen}}|} \sum_{j \in y_{\text{gen}}} \max_{i \in y_{\text{ref}}} \text{sim}(y_{\text{ref}_i}, y_{\text{gen}_j})$$

$$R = \frac{1}{|y_{\text{ref}}|} \sum_{i \in y_{\text{ref}}} \max_{j \in y_{\text{gen}}} \text{sim}(y_{\text{ref}_i}, y_{\text{gen}_j})$$

The F1 and F3 scores are then computed to balance precision and recall, emphasizing recall more heavily in F3:

$$F1 = \frac{2 \cdot P \cdot R}{P + R}$$

$$F3 = \frac{10 \cdot P \cdot R}{9 \cdot P + R}$$

Here, y_{ref} and y_{gen} represent tokens from the reference and generated code, respectively, and sim denotes the cosine similarity between their embeddings produced by CodeBERT [23].

Justification for Choice:
This metric is favored due to its enhanced capability to recognize the quality of code beyond mere lexical similarity, capturing semantic equivalence even when lexical forms differ. The higher correlation with human judgment ratifies its efficacy in practical applications where functional accuracy is paramount.

Example of How It Works:
If a machine generates a code snippet `int add(int a, int b) {return a+b;}` and the reference code is `int add(int x, int y) {return x+y;}`, CodeBERTScore would analyze the semantic similarity of these snippets beyond mere lexical matching, evaluating whether variables and operations align semantically.

These metrics, BLEU scores and CodeBERTScore, therefore, provide a robust framework for evaluating the performance of our models, ensuring both the fidelity and the utility of the generated outputs in real-world applications.

Model Training Mode and Loop

```
# Model Training Mode
transformer.train()

# Training Loop
for epoch in range(100):
optimizer.zero_grad()
output = transformer(src_data, tgt_data[:, :-1])
loss = criterion(output.contiguous().view(-1, tgt_vocab_size)
 , tgt_data[:, 1:].contiguous().view(-1))
loss.backward()
optimizer.step()
print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

Monitoring Progress

The training progress is monitored through the printing of the epoch number and loss value for each epoch Lightning Module.

Chapter 5

Results and Discussions

5.1 Results

Inference:

```
Prompt:
Program to find value of f(n) = k+2+k+3+k+...+n, k=1
Find the value of pow (i, 4)
and then add it to the sum;
Return the sum;

Generated code:
def sumOfSeries(n):
sum = 0
for i in range(1, n+1):
sum += (i*k)
return sum
def sumOfSeries(n):
sum = 0
sum = 0

if __name__ == "__main__":
K=1
print(sumOfSeries(sumOfSeries(n)))
```

Listing 5.1: Sample code generated by the model in response to a prompt

The model exhibits proficiency in understanding the prompt and translating it into a syntactically correct Python function. As shown in the generated code, the model successfully defines a function sumOfSeries that computes the sum of a series according to the given mathematical formula.

However, upon closer examination of the generated code, we observe a discrepancy in the quality of output between different parts of the code. The first part

of the generated code accurately follows the prompt by initiating a sum to zero and implementing a loop that computes the series' sum. The logic encapsulated here demonstrates the model's capability to adhere to the structured sequence of operations needed to solve the problem.

In contrast, the second part of the code, although syntactically correct, repeats the function definition and contains redundant statements which do not align with Pythonic practices or the original prompt's requirements. Furthermore, the final print statement attempts to pass the function sumOfSeries as an argument to itself, which reflects a misunderstanding of the function's purpose and indicates a clear divergence from the expected logic.

This qualitative assessment raises important points about the model's current limitations. While it is capable of generating coherent and relevant code segments, it can sometimes produce redundant or even erroneous sequences. Such observations underscore the necessity of further refining the model's capabilities, especially in terms of code understanding and logical consistency within larger blocks of code.

Model	BLEU	F1	F3	Precision	Recall
XLCoST	15.09	0.76	0.74	0.78	0.75
Alpaca	3.54	0.72	0.72	0.72	0.72
Merged	10.87	0.70	0.69	0.72	0.68

Table 5.1: Model Performance Evaluation Metrics

The primary metric for evaluating the quality of machine-generated code in comparison to source code was the BLEU score, complemented by the CodeBERTScore. Despite BLEU being traditionally used for such assessments, its reliance on n-gram matching can lead to lower scores even when variable names or syntax vary but functional correctness is retained. For the model results presented (see Table 1), the BLEU score suggests a moderate level of similarity to reference implementations, with XLCoST at 15.09 and Alpaca at 3.54 out of a possible 100. This reflects the stringent nature of the BLEU metric, which may not fully capture semantic correctness or human preference in code generation tasks.

In contrast, the CodeBERTScore presents a nuanced picture. Designed to evaluate code by considering the consistency between the generated code and its given natural language context, CodeBERTScore was found to correlate better with human preferences and functional correctness of generated code than BLEU and other existing metrics [23]. This indicates that CodeBERTScore could provide a more accurate assessment of a model's ability to produce functionally valid and human-like code.

5.2 Discussions

The relatively low BLEU scores, when considered in isolation, might be misleading [23]. In the domain of code generation, a BLEU score that may be considered low in machine translation can still signify a reasonable level of functional correctness. For example, the XLCoST model's BLEU score of 15.09, while modest, does not necessarily indicate poor performance. For comparison, large language models like GPT-3 have demonstrated BLEU scores of 35.6 [24]. It is essential to contextualize these scores within the domain-specific challenges of code generation, where different implementations can achieve the same functionality.

Moreover, the paper by Zhou et al. (2023) on CodeBERTScore provides valuable insights into the limitations of BLEU for code evaluation [23]. While BLEU focuses on exact lexical matches, which is insufficient for assessing the diversity in code implementation, CodeBERTScore accommodates the variability in naming conventions and coding styles, thus being more aligned with the practical requirements of code evaluation.

The XLCoST model's performance, though lower in BLEU of 15.09, may still be competent in terms of human preferences and functional correctness.

In conclusion, while BLEU offers a foundational measure of text generation accuracy, CodeBERTScore provides a more comprehensive evaluation framework, particularly suited to the domain of code generation [25]. Future work and model assessments could benefit from incorporating both metrics, with a greater emphasis on CodeBERTScore for insights into functional and human-aligned code generation capabilities.

Chapter 6

Conclusion

The results of our research affirm the capability of the designed deep learning model to generate code effectively. Despite achieving a relatively low BLEU score of 15.09, 2.51, 10.87 for three datasets, the significance of this metric must be considered in context. The BLEU score, commonly used as a standard measure for comparing machine-generated text to human-written references, may not always reflect the true semantic and functional accuracy of generated code. For instance, GPT models, known for their robust performance, typically achieve a BLEU score of around 35.6, suggesting that even high-performing models do not reach perfect scores, thus questioning the reliability of BLEU in evaluating code generation [24]. In contrast, CodeBERTScore, a more recent metric developed specifically for code evaluation, offers a more nuanced assessment by considering semantic similarity and the functional role of variables in the code. Our results, which demonstrated a much higher CodeBERTScore, affirm that this metric correlates more strongly with human preferences and the functional correctness of the code . This correlation is pivotal as it aligns more closely with the practical requirements of code generation, where the ability of the code to perform its intended function is more critical than syntactic exactness. The quantitative measures from our evaluation highlight this distinction: while the BLEU score was modest, the CodeBERTScore was significantly higher, reflecting a deeper alignment with the intended functionality and readability of the code as evaluated by human standards. These findings justify our confidence in the model's output, underscoring its effectiveness and efficiency in generating usable code. However, the findings suggest that while the model shows promise, it may benefit from additional fine-tuning, particularly in parsing complex nested operations and maintaining context throughout longer stretches of code. This will ensure that each part of the generated code not only stands correct independently but also contributes correctly to the overall functionality as intended by the input prompt. For future research, one promising direction would be the integration of more advanced architectures such as GPT-2 or its succes-

sors as the decoder component in the model. This could potentially enhance the model's understanding and generation capabilities, leading to even higher quality code output. Our results lay a solid foundation for this exploration, suggesting that such advancements could further elevate the performance and utility of code generation models.

Bibliography

[1] Yuhang Lai et al. "DS-1000: A natural and reliable benchmark for data science code generation". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 18319–18345.

[2] Enrique Dehaerne et al. "Code generation using machine learning: A systematic review". In: *Ieee Access* (2022).

[3] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[4] Saidul Islam et al. "A comprehensive survey on applications of transformers for deep learning tasks". In: *Expert Systems with Applications* (2023), p. 122666.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[6] Partha Pratim Ray. "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope". In: *Internet of Things and Cyber-Physical Systems* (2023).

[7] Yujia Li et al. "Competition-level code generation with alphacode". In: *Science* 378.6624 (2022), pp. 1092–1097.

[8] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[9] Yao Wan et al. "Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit". In: *arXiv preprint arXiv:2401.00288* (2023).

[10] Ming Zhu et al. "Xlcost: A benchmark dataset for cross-lingual code intelligence". In: *arXiv preprint arXiv:2206.08474* (2022).

[11] Giovanni Pinna et al. "Enhancing Large Language Models-Based Code Generation by Leveraging Genetic Improvement". In: *European Conference on Genetic Programming (Part of EvoStar)*. Springer. 2024, pp. 108–124.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27 (2014).

[13] Maryville University. "Top 4 Data Analysis Techniques That Create Business Value". In: (2023), https://online.maryville.edu/blog/data~analysis~techniques/ [Accessed: (2023–08–20)].

[14] Uri Alon et al. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[15] Khajamoinuddin Syed et al. "Integrated natural language processing and machine learning models for standardizing radiotherapy structure names". In: *Healthcare*. Vol. 8. 2. MDPI. 2020, p. 120.

[16] Baptiste Roziere et al. "Code llama: Open foundation models for code". In: *arXiv preprint arXiv:2308.12950* (2023).

[17] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[18] Berdakh Abibullaev, Aigerim Keutayeva, and Amin Zollanvari. "Deep Learning in EEG-Based BCIs: A Comprehensive Review of Transformer Models, Advantages, Challenges, and Applications". In: *IEEE Access* (2023).

[19] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[21] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[22] Kishore Papineni et al. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[23] Shuyan Zhou et al. "Codebertscore: Evaluating code generation with pretrained models of code". In: *arXiv preprint arXiv:2302.05527* (2023).

[24] Aishwarya Narasimhan, Krishna Prasad Agara Venkatesha Rao, et al. "Cgems: A metric model for automatic code generation using gpt-3". In: *arXiv preprint arXiv:2108.10168* (2021).

[25] Terry Yue Zhuo. "Large language models are state-of-the-art evaluators of code generation". In: *arXiv preprint arXiv:2304.14317* (2023).