

Curious Learner: A Generative Neuro-Symbolic Approach for Function Execution & Illustration Using Natural Language

by

A.F.M. Mohimenul Joaa
21166040

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
M.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
February 2024

© 2024. Brac University
All rights reserved.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

A handwritten signature in black ink that reads "Mohimenuul". The signature is written in a cursive style and is centered within a light blue rectangular box.

A.F.M. Mohimenuul Joaa

21166040
Student ID

Approval

The thesis titled “Curious Learner: A Generative Neuro-Symbolic approach for function Execution & Illustration using Natural Language” submitted by

1. A.F.M. Mohimenul Joaa (21166040)

Of Spring, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of M.Sc. in Computer Science on February 21, 2024.

Examining Committee:

Supervisor:

(Member)



Dr. Farig Yousuf Sadeque

Assistant Professor
Department of Computer Science and Engineering
BRAC University

Internal Examiner:

(Member)

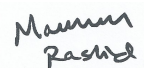


Dr. Md. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
Brac University

External Examiner:

(Member)



Dr. Mamunur Rashid

Assistant Professor
Computational Biology and Bioinformatics
University of Birmingham

Head of Department:

(Chair)



Dr. Sadia Hamid Kazi

Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

M.Sc. Thesis Coordinator:



Dr. Md Sadek Ferdous

Associate Professor
Department of Computer Science and Engineering
Brac University

Ethics Statement

Title: Curious Learner: A Generative Neuro-Symbolic Approach for Function Execution & Illustration Using Natural Language.

Author: A.F.M. Mohimenul Joaa

Supervisor: Dr. Farig Yousuf Sadeque

Department: Department of Computer Science and Engineering

University: BRAC University

1. Ethical Approval:

This thesis received approval from the aforementioned examining Committee to ensure adherence to ethical guidelines.

2. Data Management:

All data used in this project is created manually, in compliance with data protection regulations, stored securely, and retained according to institutional policies.

Abstract

Generative models possess immense potential, but their ability to perform complex calculations is limited by the need to memorize vast amounts of data, leading to computational inefficiencies. Leveraging tools like the Arithmetic Logic Unit using symbolic functions offers a more efficient alternative, enabling faster responses, smaller model sizes, and improved accuracy. We propose a neuro-symbolic generative model to empower natural language models with task execution abilities by integrating functional programming principles. Experiments on our scoped four translation tasks using 98 mathematical functions demonstrated rapid convergence and minimal training time requirements. Our model, containing 111 million trainable parameters, achieved an average accuracy, BLEU score, and perplexity score of 0.85, 0.84, and 5.9, respectively, after training on a T4 GPU for several hours. This neuro-symbolic Language Model shows significant potential for various applications, such as NLP-based command line tools, customer service automation, service discovery automation, project code automation, and natural language-based operating systems.

Keywords: Curious Learner; Transformer; Foundational Model; Natural Language Processing; Generative Model; Neuro-Symbolic Programming; Large Language Model Architecture; Task Executor; Customer Service Automation; Service Discovery Automation

Acknowledgement

All praise to the Great Allah for whom our thesis have been completed without any major interruption.

My Supervisor Dr. Farig Yousuf Sadeque sir for his kind support and advice in our work. He helped us whenever we needed help.

My Friend Prattoy Majumder for his support, motivation and contribution in the project and thesis.

The whole examining committee of the thesis.

And finally to my family without their throughout sup-port it may not be possible.

Table of Contents

Declaration	i
Approval	ii
Ethics Statement	iv
Abstract	v
Acknowledgment	vi
Table of Contents	vii
List of Figures	x
List of Tables	xi
Nomenclature	xii
1 Introduction	1
1.1 Preface	1
1.2 Scope of the Study	2
1.3 Research Design	3
1.4 Structure of the Thesis	3
2 Literature Review	5
2.1 Historical Background	5
2.2 Neurosymbolic Generative Models	5
2.3 Synthesis of Existing Studies	5
2.3.1 DreamCoder	6
2.3.2 ChatGPT Plugin	6
2.3.3 ChatGPT Function Calling	6
2.3.4 Rasa	6
2.4 Critique of Existing Literature	6
2.5 Future Directions	7
3 Data	8
3.1 Overview	8
3.2 Functions	11
3.3 Tasks	11
3.3.1 Function to Function Translation	11

3.3.2	Function to NL Translation	11
3.3.3	NL to Function Translation	11
3.3.4	NL & Function to NL & Function Translation	15
4	Methodology	16
4.1	Approach Overview	16
4.2	Vanilla Transformer	16
4.3	Curious Learner Architecture Selection	17
4.3.1	Architecture Diagram	18
4.3.2	Input Parser	18
4.3.3	Tokenizer	20
4.3.4	ALiBiBi-Attention with Linear Bidirectional Biases Encoder	20
4.3.5	Category and Task Encoder	21
4.3.6	Common Block	22
4.3.7	Category Map Block	23
4.3.8	Category Map Decoder	23
4.3.9	Category Map Classification Head	23
4.3.10	Category Router	24
4.3.11	Output Token Block	24
4.3.12	Output Token Decoder	24
4.3.13	Output Token Classification Head	25
4.3.14	Response Parser	26
4.4	Architecture Justification	26
4.5	Building Vocabulary	27
4.5.1	Category Map Vocabulary Builder	28
4.5.2	Output Token Vocabulary Builder	28
4.6	Different Types of Embeddings	28
4.6.1	Token Embeddings	29
4.6.2	ALIBIBI Embeddings	29
4.6.3	Category and Task Embeddings	29
4.6.4	Combined Embeddings	30
4.7	Saving, Loading, and Retraining the Model	30
4.8	Data Loader	31
4.8.1	Data Generator	31
4.8.2	Batch Builder	31
4.9	Training and Inference Methods	32
4.9.1	Guiding Tokens to Output Token Classification Heads	33
4.9.2	Training and Inference Types	33
4.9.3	Training Process	33
5	Result	35
5.1	Hyperparameter Selection	35
5.1.1	Activation Function	35
5.1.2	Normalization	36
5.1.3	Regularization	36
5.1.4	Learning Rate Scheduler	36
5.1.5	Optimizer	37
5.1.6	Criterion	37
5.1.7	Epoch	37

5.1.8	Training Batch Size	38
5.1.9	Number of Heads in Attention Layer	38
5.1.10	Number of Layers in Decoder	38
5.1.11	Hidden Embeddings Dimension	38
5.1.12	Feed Forward Layer Dimension	39
5.1.13	Max Decoding Length	39
5.1.14	Add BOS and EOS Tokens	39
5.1.15	Data Loader Parameters	39
5.2	Acuracy Increment per Epoch	40
5.3	Loss Decrement per Epoch	42
5.4	Inference	42
6	Discussion	46
6.1	Challenges and Limitations	46
6.2	Future Works	46
6.3	Conclusion	47
	Bibliography	50

List of Figures

4.1	Vanilla Transformer Architecture Diagram	17
4.2	Curious Learner Model Architecture Diagram	19
4.3	Category and Task Embeddings Signal	22
4.4	Category Router Detailed Architecture Diagram	25
4.5	Hub vs Switch category routing	27
4.6	ALiBiBi Bias for All Attention Heads	29
4.7	Non-generative and Generative Training	34
5.1	Learning Rate Scheduler Graph[15]	36
5.2	Curious Learner Learning Rate Scheduler Graph	37
5.3	Average Accuracy vs Epoch Graph	40
5.4	Accuracy vs Epoch Graph	41
5.5	Average Accuracy vs Epoch Graph	42
5.6	Loss vs Epoch Graph	43
5.7	Parsed Result Example for Each Task	44
5.8	Inference Evaluation Metrics vs Epoch Graph	45

List of Tables

3.1	Category Map Detail	9
3.2	Function Prefixes and Their Meanings	9
3.3	Mathematical Function Signatures and Return Types(0-42)	12
3.4	Mathematical Function Signatures and Return Types(43-85)	13
3.5	Mathematical Function Signatures and Return Types(86-97)	14
3.6	Function to Function Translation Example	14
3.7	Function to NL Translation Example	14
3.8	NL to Function Translation Example	14
3.9	NL & Function to NL & Function Translation Example	14
5.1	Performance Metrics	43

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

AGI Artificial General Intelligence

ALIBIBI Attention with Linear Bidirectional Biases

BERT Bidirectional Encoder Representations from Transformers

BLEU Bilingual Evaluation Understudy

CL Curious Learner

E1 Embedding for Category

E2 Embedding for Output Token

GPT Generative Pre-trained Transformer

IP Input Parser

IPO Input Parser Output

LLM Large Language Model

NLP Natural Language Processing

RP Response Parser

XLNet eXtreme Learning NETwork

Chapter 1

Introduction

1.1 Preface

ChatGPT and other Large language models have shown how scaling transformer models can be useful for learning, reasoning, and querying any specific knowledge. Standing on this discovery many are trying to build artificial general intelligence. Everyone has their approach to solving the AGI problem. Some are creating agents like AutoGPT[36], which can recursively call LLM to figure out a workflow, and then solve the work by again asking to solve the steps. Some are creating tools for fine-tuning currently available models on specific domains. Some are trying to make multi-modal models.

Our proposition hinges on the observation that as humans we solve problems not only by memorizing but also by using different tools, which we know can do our task. We learn these tools and use them for accurate answers rather than reinventing the wheel every time. So if we can give access to these tools to an LLM then it can do so many tasks for us. ChatGPT is solving this issue by using plugins[31], but it's an add-on service where the model is given some extra prompts to accommodate the function calling of the plugins. Similarly, for getting structural data from ChatGPT using the function calling[32] feature, it uses its current capability of understanding the language to identify parameters and return them in a structural format. So as of now, there is no scalable solution that can give LLM capabilities to use tools like we humans do.

We want to address this issue, by creating NLP based model that can identify proper symbolic functions from prompt and execute them by collecting proper parameters from the user. So our main idea is to add functional programming principles with LLM to make a scalable Neuro-Symbolic generative model that can use tools just like us to solve problems. If successfully scaled, our proposed solution could be effectively employed in various applications:

- Interactive chat UI with dynamic model response and user input widgets.
- Customer service and service discovery chatbots.
- Natural Language-based Command Line Tools (e.g., GitHub CLI, npm, Azure CLI), enhancing user interaction with systems.

- Model reliability: The correctness of the model’s output hinges on its ability to identify the correct function.
- Natural Language-based Operating System.

We start from what we know. As transformer models are working great for NLP-related tasks we started from a vanilla transformer model. The idea was Instead of the word as a token what if we explore the concept of utilizing a map structure with metadata, encompassing diverse token types such as words, functions, integers, lists, etc. While the metadata includes its type, subtype, and subsubtype. However, this approach poses challenges, including the inability to utilize conventional tokenizers and the exponential increase in token count due to unique token-metadata combinations. Moreover, vanilla transformers can’t work with words and sentences at the same time. It can’t ensure the proper order and type of param for function execution. Also due to sinusoidal positional embeddings no additional information can be embedded along with embeddings.

So to mitigate these issues we add components gradually. For extracting token metadata information from the raw string as well as to execute functions we add the input parser. However, since humans may not understand these technicalities, we add the response parser at the output end to adjust the tokens for human comprehension. As functions are best represented by multiple sentences using doc_string, we used the sentence transformer to get the initial vector embedding of functions. We need to ensure the function parameter type and order to execute a function properly, which we are calling hard constraints. To ensure the hard constraints we are predicting the category of a token first before predicting the real token. Then using this category we are routing embedding through expert classification heads to retrieve the specific type of token. We utilized Attention with Linear Bidirectional Bias instead of sinusoidal positional embedding to enhance the integration of additional information with tokens. We introduce the category and task encoder to incorporate category and task information into token embeddings and utilize cross-attention to the function signature to discern parameter types and order. Consequently, our architecture diverges significantly from the vanilla transformer, leading us to term it "Curious Learner."

1.2 Scope of the Study

As we all know, training a foundational model is computationally demanding, and for independent researchers, accessing such computational power is often not feasible. Therefore, we have deliberately limited the scope of our research to focus on a specific subset of tasks and data.

To this end, we have identified 98 mathematical functions and developed training and inference data generators using these functions. Further details about these functions can be found in the data chapter of the thesis. Given the mammoth task of creating a generative large language model (LLM), which requires substantial computational power and engineering effort, it is also imperative to scope the tasks on which the model will be trained and tested. We have identified four tasks that scope the training and testing scope, as well as our computational requirements:

- i. Function to Function Translation Dataset
- ii. Function to Natural Language Translation Dataset
- iii. Natural Language to Function Translation Dataset
- iv. Natural Language & Function to Natural Language & Function Translation Dataset

Since our dataset and task are unique and no similar dataset is available, we can't conduct any comparative study. This architecture can be further scaled if we can implement the switch routing mechanism and create categories based on function signatures rather than predicting function parameter categories, which are the bottleneck for computation power. So with them solved along with a larger dataset our model can outperform our current presented evaluation with a big margin.

1.3 Research Design

Our research design encompasses the development and evaluation of two distinct models: a vanilla transformer model and a decoder-only generative curious learner model. Initially, we constructed a vanilla transformer model utilizing a custom vocabulary. In this vocabulary, each unique category map and token combination was considered a token. The specifics of this model, including architecture and training methodology, are detailed in the methodology chapter. Subsequently, we developed our decoder-only generative curious learner model. This model architecture, which incorporates functions as tokens to determine the appropriate function to execute and its parameters, is thoroughly explained in the methodology chapter. To ensure the generated tokens adhere to the correct order and type of parameter of the functions, we adopted a two-step approach. Firstly, we predicted the category of the token. Then, based on the predicted category, we employed a specific output classification head to predict the token itself. Additionally, we implemented an Input Parser to facilitate processing of natural language data. Similarly, to convert predicted categories and tokens into human-readable natural language, we developed a Response Parser.

1.4 Structure of the Thesis

This thesis is structured into Six chapters:

- i. **Introduction:** This chapter provides an overview of the research topic, motivations, objectives, and outlines the structure of the thesis.
- ii. **Literature Review:** The second chapter reviews relevant literature and existing research in the field of neuro-symbolic generative foundational models in natural language processing.
- iii. **Data:** Chapter three discusses the data utilized in the study, including the selection, preprocessing, and creation of datasets.

- iv. **Methodology:** In chapter four, the methodology employed in developing and evaluating the models, including the architecture, training procedures, and evaluation metrics, is detailed.
- v. **Results:** Chapter five presents the results obtained from the experiments conducted with the developed models.
- vi. **Discussion:** Chapter six provides a comprehensive discussion of the findings, interpretations, and implications of the results, as well as limitations and future research directions.

Chapter 2

Literature Review

2.1 Historical Background

Neuro-symbolic programming represents a convergence of neural network-based approaches and symbolic reasoning, aiming to leverage the strengths of both paradigms in AI systems[28]. This interdisciplinary field traces its origins back to early AI research, where the idea of combining neural and symbolic methods emerged as researchers sought to bridge the gap between logic-based abstraction and pattern recognition from data[27], [28]. In the early stages, significant efforts were made to integrate handwritten symbolic rules into neural networks, refining them with data to improve generalization and efficiency[28]. Over time, research evolved to extract symbolic models from neural networks, demonstrating enhanced performance compared to traditional neural networks[27]. This convergence of neural and symbolic approaches was further developed in frameworks such as the Connectionist Inductive Learning and Logic Programming system[28]. Recent studies have explored employing large language models (LLMs) to synthesize programs from natural language descriptions, potentially revolutionizing program synthesis methods[28]. These LLMs, such as GPT and BERT, have demonstrated remarkable capabilities in language understanding and generation tasks[14], [21], [26].

2.2 Neurosymbolic Generative Models

Neurosymbolic generative models combine neural and symbolic techniques to generate data adhering to specific constraints or patterns[18]. They leverage neural networks and symbolic reasoning, demonstrating promising capabilities in tasks like program synthesis[27] and programmatic expression generation[24].

2.3 Synthesis of Existing Studies

The field of neurosymbolic computing encompasses a wide range of research, from foundational theory to practical applications. Early work focused on theory and basic implementations[3], but recent advances have led to more sophisticated models[28]. Integration with other AI paradigms, like reinforcement learning and program synthesis, has broadened its applicability[22]. Yet, challenges such as scalability and interpretability persist[7].

2.3.1 DreamCoder

DreamCoder advances neurosymbolic programming by automatically generating interpretable programs through Bayesian methods[22]. This facilitates learning complex tasks with limited data, aiding in generalization to new scenarios.

2.3.2 ChatGPT Plugin

ChatGPT have implemented initial support for plugins on March 23, 2023[31]. Plugins are tools designed specifically for language models with safety as a core principle, and help ChatGPT access up-to-date information, run computations, or use third-party services. By combining neural language understanding with symbolic manipulation of dialogue context, ChatGPT-Plugin enhances conversational agents' ability to generate coherent and relevant responses in various domains[33].

2.3.3 ChatGPT Function Calling

ChatGPT announced updates including more steerable API models, function calling capabilities, longer context, and lower prices on 13 June 2023[32]. Since the alpha release of ChatGPT plugins, significant advancements have been made in integrating tools and language models securely. Function calling allows developers to more reliably get structured data back from the model. Developers can consume information from trusted tools and by perform actions on users behalf with real-world impact, such as sending an email, posting online, or making a purchase.

2.3.4 Rasa

Rasa, facilitates the development of contextual assistants capable of layered conversations with extensive back-and-forth interactions[8]. Unlike our approach, which utilizes an attention layer to execute functions with proper parameters in a single shot, Rasa identifies the user's intent first and then asks further questions to execute a task based on the intent. This distinction highlights the capability of our approach to execute functions using proper parameters without extensive back-and-forth interactions efficiently.

2.4 Critique of Existing Literature

Despite extensive research in neurosymbolic computing and impressive advancements in various applications, a notable gap persists in executing functions using large language models (LLMs). While studies have explored LLMs' capabilities for language tasks[14], [21], [26], none systematically tackle function execution from natural language prompts[28]. This underscores the need for novel approaches combining LLMs' power with symbolic reasoning to execute functions directly from natural language. While ChatGPT's function calling capabilities have seen considerable progress in its plugin functionality, there remains a notable absence of information regarding its architecture. Moreover, it appears that the design of ChatGPT has not been explicitly tailored to adhere to functional programming principles. Consequently, there exist opportunities for exploring novel architectures that could

better accommodate function calling with large language models (LLMs). Such architectures could potentially address existing gaps and enhance the efficiency and effectiveness of function calling within the context of LLMs.

2.5 Future Directions

Our research identifies a gap in function execution using LLMs, offering a promising avenue for future investigation. Future research should focus on optimizing the integration of LLMs with symbolic reasoning techniques to enhance function execution efficiency and accuracy[26]. Additionally, exploring LLM-based function execution across various domains is crucial, including software development and data analysis[18]. By evaluating performance and scalability in practical settings, researchers can demonstrate utility and identify areas for improvement, advancing neurosymbolic computing in automating tasks and improving human computer interaction.

Chapter 3

Data

3.1 Overview

We want to use functions as tokens of the generative LLM, so we searched the web for this type of dataset, but quickly we figure out that, we need full customization capability of our data, as our model architecture is new. We looked into the following dataset.

1. BigCloneBenc : The BigCloneBench dataset is a significant resource in software engineering for code clone detection research. It comprises over pairs of code clones extracted from diverse open-source projects in various programming languages like Java, C, and C++. These clones encompass different types, including identical, syntactically similar, and semantically similar clones. Each clone pair is manually annotated with clone type and similarity information. The dataset serves as a benchmark for evaluating the effectiveness of clone detection algorithms and is freely available for research purposes, aiding in the advancement of code clone detection techniques.

But none of them fulfill our requirements. At last we build our own dataset. Initially we create a set of 98 mathematical functions, for using them as the function tokens. Normally we will provide only text to the model, model will parse the input and create (value, category map) from each token. We have created a input parser for this purpose. Category map have three information, each of them are enum which can take values from their respective sub lists. In Table: 3.1, we provided the values of these enums.

As our model will predict the category map of the next token first. Then based on the category map it will router the output token embeddings through the predicted output token classification head to predict the output token. So to combine these information and present the result to the end user, we build a response parser. The output of response parser is human readable code or natural language or both. Whereas the input of the input parser need some modification for capturing the function information.

Functions can be represented using four different prefixes. In Table: 3.2, we provided the meanings of these function prefixes.

Some example triplet of model input(Input Parser input), Input parser Output (Response Parser input) & Response Parser output is provided below.

CategoryType	CategorySubType	CategorySubSubType
function	return_value	none
word	default	placeholder
integer	integer	param_one
float	float	param_two
list	list	param_three
bool	bool	para_four
special	word	param_five
	placeholder	param_last
		execute
		represent

Table 3.1: Category Map Detail

Prefix	Meaning
\$\$	Input Parser(IP) Execute
##	Input Parser(IP) Represent Response Parser(RP) Execute
@@	Input Parser(IP) and Response Parser(RP) Placeholder
&&	Input Parser(IP) and Response Parser(RP) Represent

Table 3.2: Function Prefixes and Their Meanings

1. Input Parser(IP) Execute

- **Model Input:** `$$add(3,4)`
- **Input Parser Output:**

```
[
  (
    7,
    {category : "integer", subCategory : "return_value", subSubCategory : "none"}
  ),
]
```
- **Response Parser Output:** 7

2. Input Parser(IP) Represent Response Parser(RP) Execute

- **Model Input:** adding 3 plus 4 = `##add(3,4)`
- **Input Parser Output:**

```
[
  (
    adding,
    {category : "word", subCategory : "default", subSubCategory : "none"}
  ),
  (
    3,
    {category : "integer", subCategory : "default", subSubCategory : "none"}
  ),
  (
    plus,
    {category : "word", subCategory : "default", subSubCategory : "none"}
  ),
  (
    4,
    {category : "integer", subCategory : "default", subSubCategory : "none"}
  ),
  (
    =,
    {category : "word", subCategory : "default", subSubCategory : "none"}
  ),
  (
    add(),
    {category : "function", subCategory : "integer", subSubCategory : "execute"}
  ),
  (
    3,
    {category : "integer", subCategory : "default", subSubCategory : "param_one"}
  ),
  (
    4,
    {category : "integer", subCategory : "default", subSubCategory : "param_last"}
  ),
]
```

- **Response Parser Output:** adding 3 plus 4 = 7

3. Input Parser(IP) and Response Parser(RP) Placeholder

- **Model Input:** @@avg(@list)

- **Input Parser Output:**

```
[
  (
    avg(),
    {category : "function", subCategory : "float", subSubCategory : "placeholder"}
  ),
  (
    @list,
    {category : "list", subCategory : "placeholder", subSubCategory : "param_last"}
  ),
]
```

- **Response Parser Output:** avg(@list)

4. Input Parser(IP) and Response Parser(RP) Represent

- **Model Input:** $\&\&avg([1, 2, 3])$
- **Input Parser Output:**

```
[  
  (  
    avg(),  
    {category : "function", subCategory : "list", subSubCategory : "represent"}  
  ),  
  (  
    [1, 2, 3],  
    {category : "list", subCategory : "placeholder", subSubCategory : "param_last"}  
  ),  
]
```
- **Response Parser Output:** $avg([1, 2, 3])$

Further detailed functionality of input parser and response parser is provided in the methodology chapter.

3.2 Functions

We have chosen the following 98 mathematical functions for practically evaluating our model.

3.3 Tasks

Our generative model can mark different task using frequency and amplitude modulation technique, more about this technique can be found in the methodology chapter. We are using the following four tasks for evaluating the model performance regarding identifying tasks,

3.3.1 Function to Function Translation

This task involves translating a given function expression into another function expression. It assesses the model's ability to understand and transform mathematical operations. Example is presented in Table 3.6.

3.3.2 Function to NL Translation

In this task, we translate a function expression into a natural language description of the calculation. It evaluates the model's capacity to generate human-readable descriptions of mathematical operations. Example is presented in Table 3.7.

3.3.3 NL to Function Translation

This task involves translating a natural language query into a corresponding function expression. It tests the model's ability to interpret and convert human language instructions into executable mathematical operations. Please refer to Table 3.8 for example pair.

Function Signature	Return Type
addition(x: int, y: int)	int
subtraction(x: int, y: int)	int
multiplication(x: float, y: float)	float
division(x: float, y: float)	float
exponentiation(x: float, y: float)	float
square_root(x: float)	float
floor_division(x: int, y: int)	int
modulus(x: int, y: int)	int
logarithm(x: float, base: float)	float
sine(x: float)	float
cosine(x: float)	float
tangent(x: float)	float
arcsine(x: float)	float
arccosine(x: float)	float
arctangent(x: float)	float
hyperbolic_sine(x: float)	float
hyperbolic_cosine(x: float)	float
hyperbolic_tangent(x: float)	float
logarithm_base_10(x: float)	float
logarithm_base_2(x: float)	float
degrees_to_radians(x: float)	float
radians_to_degrees(x: float)	float
gcd(x: int, y: int)	int
lcm(x: int, y: int)	int
isqrt(x: int)	int
pow_mod(x: int, y: int, mod: int)	int
ceil(x: float)	int
floor(x: float)	int
round(x: float)	int
absolute_difference(x: float, y: float)	float
greatest_value(x: float, y: float)	float
smallest_value(x: float, y: float)	float
product(numbers: list)	float
factorial(x: int)	int
is_prime(x: int)	bool
prime_factors(x: int)	list
is_perfect_square(x: int)	bool
is_perfect_cube(x: int)	bool
mean(numbers: list)	float
median(numbers: list)	float
relu(x: float)	float
ascending_sort(lst: list[int])	list[int]
descending_sort(lst: list[int])	list[int]

Table 3.3: Mathematical Function Signatures and Return Types(0-42)

Function Signature	Return Type
square_int(x: int)	int
square(x: float)	float
absolute(x: float)	float
power_of_ten(x: float)	float
cube(x: float)	float
cube_root(x: float)	float
is_even(x: int)	bool
is_odd(x: int)	bool
max_value(lst: list[int])	list[int]
min_value(lst: list[int])	list[int]
nth_root(x: float, n: int)	float
geometric_mean(lst: list[float])	float
is_power_of_two(x: int)	bool
binary_to_decimal(binary)	int
decimal_to_binary(decimal)	str
is_palindrome(x: str)	bool
sum_of_digits(x: int)	int
hypotenuse(a: float, b: float)	float
circle_area(radius: float)	float
permutation(n: int, r: int)	int
combination(n: int, r: int)	int
invert_number(number: float)	float
float_to_int(value: float)	int
int_to_float(value: int)	float
geometric_series_sum(a: float, r: float, n: int)	float
sigmoid(x: float)	float
cosine_similarity(vector1: list, vector2: list)	float
euler_totient(n: int)	int
l1_norm(vector: list)	float
l2_norm(vector: list)	float
average(numbers: list)	float
sum(numbers: list)	float
length(numbers: list)	float
check_same_string(str1: str, str2: str)	bool
reverse_string(input_str: str)	str
get_pi()	float
get_e()	float
calculate_dot_product(vector1: list, vector2: list)	int
a_plus_b_whole_square(a: int, b: int)	int
a_squared_plus_2ab_plus_b_squared(a: int, b: int)	int
a_minus_b_whole_squared_plus_4ab(a: int, b: int)	int
a_minus_b_whole_squared(a: int, b: int)	int
a_squared_minus_2ab_plus_b_squared(a: int, b: int)	int

Table 3.4: Mathematical Function Signatures and Return Types(43-85)

Function Signature	Return Type
a_plus_b_whole_squared_minus_4ab(a: int, b: int)	int
a_squared_plus_b_squared(a: int, b: int)	int
negative_2ab(a: int, b: int)	int
positive_2ab(a: int, b: int)	int
x_plus_a_times_x_plus_b(x: int, a: int, b: int)	int
x_squared_plus_a_plus_b_times_x_plus_ab(x: int, a: int, b: int)	int
a_cubed_plus_b_cubed(a: int, b: int)	int
a_plus_b_whole_cubed_minus_3ab_times_a_plus_b(a: int, b: int)	int
a_plus_b_times_a_squared_minus_ab_plus_b_squared(a: int, b: int)	int
a_cubed_minus_b_cubed(a: int, b: int)	int
a_minus_b_whole_cubed_plus_3ab_times_a_minus_b(a: int, b: int)	int
a_minus_b_times_a_squared_plus_ab_plus_b_squared(a: int, b: int)	int

Table 3.5: Mathematical Function Signatures and Return Types(86-97)

Input	##mean([18,57,72,39])
Output	##division(##sum([18,57,72,39]),##length([18,57,72,39]))

Table 3.6: Function to Function Translation Example

Input	##addition(18,18)
Output	Calculation: 18 + 18

Table 3.7: Function to NL Translation Example

Input	Find the area of a circle with radius 90.5
Output	##circle_area(90.5)

Table 3.8: NL to Function Translation Example

Input	If you evenly distribute 54 candies among 18 children, how many candies does each child receive
Output	Each child will receive ##division(54,18) candies

Table 3.9: NL & Function to NL & Function Translation Example

3.3.4 NL & Function to NL & Function Translation

In this task, we translate a combination of natural language and function expression into another combination of natural language and function expression. It assesses the model's capability to handle complex queries involving both textual instructions and mathematical operations. Example is presented in Table 3.9

Chapter 4

Methodology

4.1 Approach Overview

In this section, we provide an overview of our approach to developing and evaluating the models used in this study. We began by creating a vanilla transformer model, utilizing a custom vocabulary that treated every unique category map and token combination as a separate token. This approach allowed us to capture fine-grained distinctions in the input data and yielded satisfactory results. Building upon the success of the vanilla transformer model, we proceeded to implement our decoder-only generative curious learner model. This model architecture, which is the focus of this chapter, incorporates functions as tokens and predict the category of the next token first and then using the that information, it selects the output classification head to predict the token itself. We find that this architecture works perfect for following the hard constraint of functional programming and functions. As function need specific order and type of params for execution, so our model can ensure that this requirement is meet. This chapter will delve into the details of both the vanilla transformer model and the curious learner architecture, including their respective training procedures, components, and design decisions.

4.2 Vanilla Transformer

In this section, we describe the development and architecture of the Vanilla Transformer model, which served as the foundation for our subsequent explorations. We utilized a custom vocabulary builder, to construct the vocabulary for the Vanilla Transformer model. This builder keep track of unique hashable *vocabItems* that consists of token, category type, category subtype and category subsubtype. The vocabulary builder includes encoding and decoding functions to map *vocabItems* to integer tokens and vice versa. The architecture of the model is the replication of the basic transformer architecture. Please refer to Figure: 4.1 for detail..

Our setup mirrors the basic Transformer architecture, featuring multi-head attention with sinusoidal positional encoding. We've added a checkpoint manager to handle model snapshots and facilitate reloading from different stages. Additionally, we've implemented a learning rate scheduler to optimize training by dynamically adjusting the learning rate. Collectively, these components form the Vanilla Transformer, streamlining training, inference, and checkpoint management.

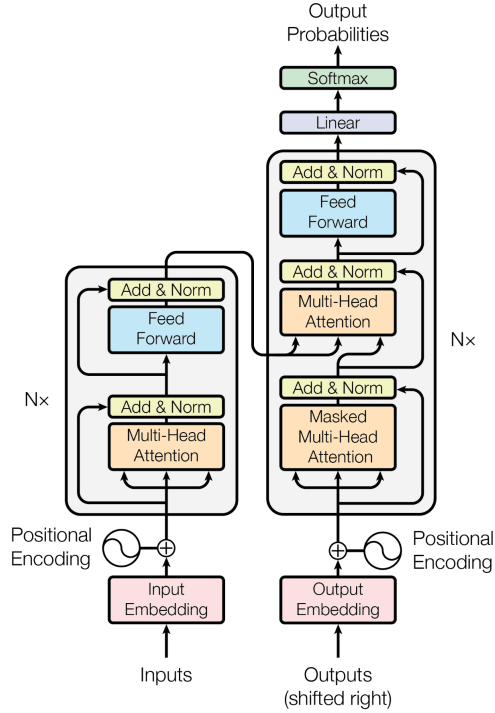


Figure 4.1: Vanilla Transformer Architecture Diagram

4.3 Curious Learner Architecture Selection

The Curious Learner represents a neuro-symbolic learning algorithm designed to execute symbolic programs based on natural language prompts. It integrates neural network components for language understanding and symbolic reasoning, enabling language models to interpret and execute symbolic functions seamlessly. The development of the Curious Learner model involved careful consideration of various factors to ensure its effectiveness in executing symbolic functions using natural language prompts. In this section, we discuss the key aspects of the architecture selection process.

In designing the architecture of the Curious Learner model, we focused on three key considerations:

1. **Hard Constraint for Function Signatures:** The model must accurately identify function signatures and strictly adhere to the order of function parameters and types. This ensures precise execution of symbolic programs, forming the foundation of the Curious Learner’s functionality.
2. **Different Tokenization Schemes for Data Types:** As different types of tokens convey distinct meanings, employing the same tokenization scheme for all data types may lead to ambiguity. Hence, we opted for separate output classification heads and tokenizers tailored to different data types, enhancing model interpretability and performance.
3. **Category and Task Information Integration:** To incorporate category and task information with the tokens, we employed frequency modulation technique. This approach facilitates the contextual understanding of input prompts and enables the model to generate appropriate responses.

4.3.1 Architecture Diagram

An architecture diagram illustrating the components and flow of information within the Curious Learner model is provided in figure: 4.2:

This diagram visually depicts the structural organization of the model and elucidates the relationships between its constituent elements. The architecture of the Curious Learner model encompasses various components, each serving a specific purpose in the execution of symbolic functions. Detailed discussions of these components, including their functionalities and interactions, are presented in the subsequent sections.

4.3.2 Input Parser

The Input Parser plays a crucial role in the Curious Learner model by converting input strings into Input Parser output. In this section, we delve into the functionality, rationale, components, and construction of the Input Parser. The Input Parser processes input text provided to the model and generates the Input Parser tuple consisting of tokens and category maps. This tuple contains information about the type, subtype, and subsubtype of each token, enabling the model to understand the structure and semantics of the input. The Input Parser was developed to handle the complexity of natural language prompts and extract relevant information for symbolic function execution. By parsing input text into categorized tokens, the model gains insights into the intended operations and parameters. It handles various input formats and extract function-related information efficiently. It employs specific prefixes to represent different types of functions. As input to the model need to have information about the function, we have introduced four prefixed to facilitate different behaviors of functions. In Table3.2, we provided the meanings of these function prefixes. Moreover, input parser categorizes tokens based on their type, subtype, and subsubtype. Each of them are enum which can take values from their respective sub lists. In Table3.1, we provided the values of these enums.

Example

Consider the following input string:

```
"##division(4.5,2)"
```

After parsing, the Input Parser generates the following IO parser output:

```
[
  (
    "<function MathFunctions.division at 0x117b828c0>",
    { "type": "function", "subType": "float", "subSubType": "execute" }
  ),
  (
    4.5,
    { "type": "float", "subType": "default", "subSubType": "param_one" }
  ),
  (
    2,
    { "type": "integer", "subType": "default", "subSubType": "param_last" },
  )
]
```

This output demonstrates how the Input Parser categorizes tokens and assigns ap-

E1 → Embedding for Category
 E2 → Embedding for Output Token

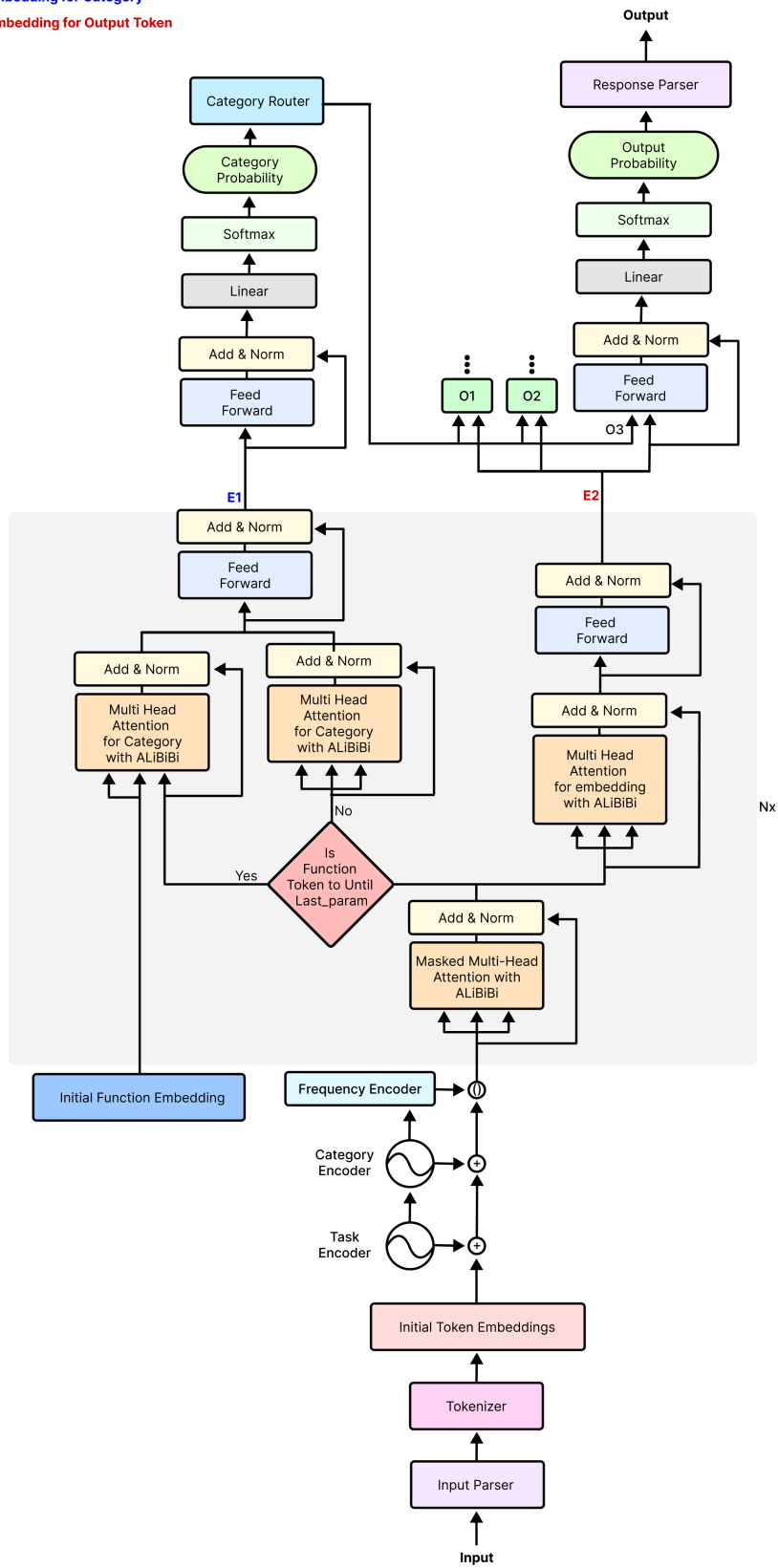


Figure 4.2: Curious Learner Model Architecture Diagram

appropriate category maps based on their characteristics, facilitating the subsequent execution of symbolic functions by the model.

4.3.3 Tokenizer

In the Curious Learner model, we utilize pre-trained models for generating initial word embeddings and function token embeddings, eliminating the need for custom tokenizers. This section discusses the approach to tokenization and embedding generation employed in the model.

Sentence Encoder[13]

We employ the "all-mpnet-base-v2" pre-trained model for generating initial word embeddings(IWE) and function docstring embeddings. Each embedding is represented as a $[1 \times 768]$ tensor, encapsulating semantic information extracted from the pre-trained model. Moreover, we experimented with our own custom vocabulary builder and tokenizer for initial word embeddings(IWE), finding that our custom tokenizer yielded superior performance, enhancing the model's accuracy

Graphcodebert Base Encoder[23]

Similarly, for function token embeddings, we utilize the "microsoft/graphcodebert-base" pre-trained model. This model generates embeddings for function tokens, capturing their semantic meaning and contextual relevance within the programming domain.

One of the primary objectives of the Curious Learner model is to accurately predict the category of the next token. To achieve this, we employ cross-attention mechanisms leveraging the signature of the function, particularly for function parameter tokens. By incorporating the function signature, the model gains contextual information about the parameters and their expected types, enhancing the accuracy of token category prediction. Additionally, we utilize the docstring of the function for generating appropriate initial tokens for functions. The docstring provides valuable contextual information about the purpose and usage of the function, aiding in the generation of meaningful initial tokens. By leveraging pre-trained models and incorporating cross-attention mechanisms along with docstring utilization, the tokenizer in the Curious Learner model ensures efficient representation and interpretation of input data.

4.3.4 ALiBiBi-Attention with Linear Bidirectional Biases Encoder

In the Curious Learner model, we opted to utilize ALiBiBi-Attention with Linear Bidirectional Biases[30] instead of traditional sinusoidal positional encoders[12], Learned Positional Embeddings[9] or Rotary positional encoder[35]. This section explores the rationale behind this decision, how ALiBiBi was integrated into the model, and the benefits it offers over positional embeddings. The decision to utilize ALiBiBi-Attention with Linear Bidirectional Biases stemmed from the need to capture longer contextual information within the model. ALiBiBi enables the model to extend its contextual understanding beyond fixed positional encodings, allowing it

to leverage bidirectional biases for more comprehensive contextual representations. This approach aligns with the model’s objective of understanding and executing complex symbolic functions based on natural language prompts.

Incorporating ALiBiBi into the Curious Learner model involved replacing traditional positional encoders with ALiBiBi-Attention mechanisms within the encoder architecture. This allowed the model to dynamically adjust attention weights based on bidirectional biases, enabling it to capture longer-range dependencies and contextual information.

ALiBiBi offers several advantages over traditional positional embeddings:

- **Longer Contextual Information:** By leveraging bidirectional biases, ALiBiBi enables the model to capture longer contextual information, facilitating a deeper understanding of input sequences.
- **Dynamic Attention Adjustment:** ALiBiBi allows for dynamic adjustment of attention weights based on bidirectional biases, enhancing the model’s ability to attend to relevant information across the input sequence.
- **Reduced Dependency on Positional Encoding:** ALiBiBi reduces the reliance on fixed positional encodings, providing more flexibility in capturing contextual relationships without explicit positional information.

Furthermore, the use of ALiBiBi facilitated the integration of the frequency modulation technique for adding task and category information with the token embeddings. By incorporating task and category information into the token embeddings, the model gains additional contextual cues, enhancing its ability to understand and execute symbolic functions accurately based on natural language prompts.

4.3.5 Category and Task Encoder

The Category and Task Encoder was implemented to enrich token embeddings with additional contextual information regarding task and category classifications. By embedding task and category information directly into token representations, the model gains enhanced contextual cues. To encode category and task information into token embeddings, we employed the frequency and amplitude modulation technique. This technique involves assigning unique signature biases for each task and category combination, which are then modulated onto the token embeddings. By modulating the token embeddings with task and category-specific frequencies and amplitudes, we ensure that each token carries contextual information relevant to its associated task and category.

Figure 4.3 illustrates the graph of category and task embeddings, token embeddings, and the amalgamated graph of category and task embeddings, alongside their corresponding fast Fourier transformed graphs.

The process of creating modulated signals involves several steps:

1. **Base Frequencies and Amplitudes:** We defined base frequencies and amplitudes for each task, categoryType, categorySubType, and categorySubSubType. These parameters serve as the foundation for modulating the token embeddings.

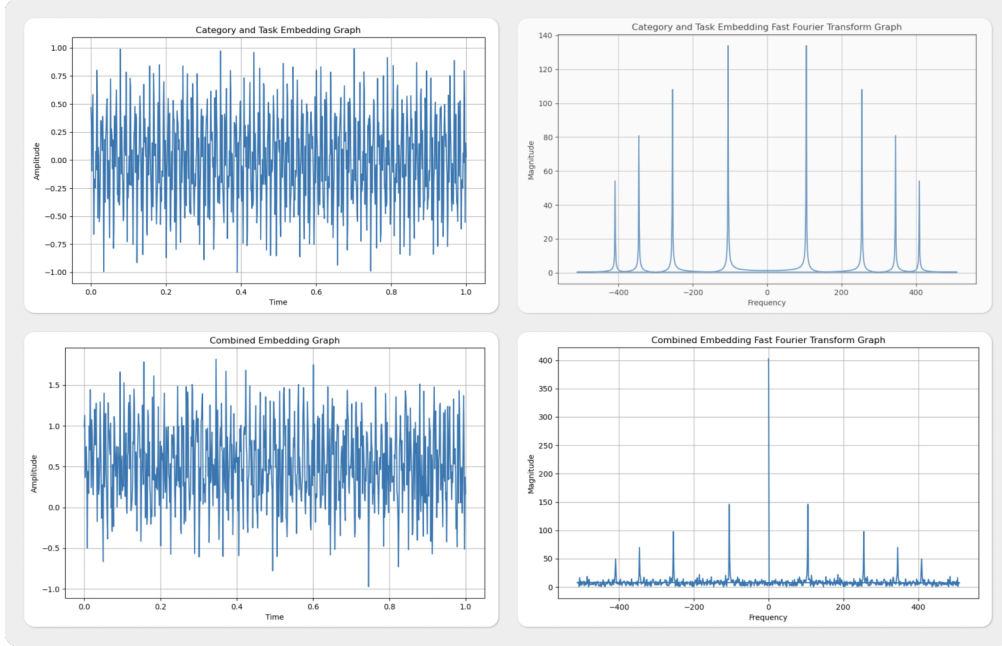


Figure 4.3: Category and Task Embeddings Signal

2. **Modulations:** We defined modulation frequencies corresponding to unique combinations of task and category information. These modulation frequencies determine how the base frequencies and amplitudes are modulated onto the token embeddings.
3. **Nyquist-Shannon Sampling Theorem:** We adhered to the Nyquist Shannon sampling theorem to determine the sampling rate of the combined signal. This ensures that the modulated signals accurately represent the encoded task and category information without aliasing or distortion.

4.3.6 Common Block

In the generative Curious Learner model, the Common Block plays a pivotal role in processing token embeddings before passing them through subsequent blocks. This section delves into the importance of the Common Block, its purpose, and the rationale behind combining both category and token information. The Common Block serves as a fundamental component within each layer of the generative Curious Learner model. Its primary purpose is to process token embeddings, ensuring that they are appropriately contextualized before proceeding to subsequent blocks. By incorporating mechanisms such as self-attention, dropout regularization, and layer normalization, the Common Block enhances the model’s ability to capture and propagate relevant information throughout the decoding process.

A key aspect of the Common Block is its capability to combine both category and token information. This integration is essential for predicting the category and token of the next output in the sequence. By jointly processing category and token embeddings, the Common Block enables the model to leverage contextual cues from both sources, enhancing the accuracy of category and token predictions.

4.3.7 Category Map Block

The Category Map Block consists of several layers designed to process token embeddings and incorporate category-specific information effectively. The block begins with a multi-head attention layer, followed by dropout regularization and layer normalization. Subsequently, a feed-forward layer is applied, followed by another round of dropout and normalization. A distinguishing feature of the Category Map Block is its token-type based attention mechanism. Depending on the current token's type, the block employs either cross attention or self-attention. Specifically, if the token represents a function or a parameter of a function (excluding the last parameter), cross attention is utilized on the signature of the function. Conversely, for other tokens, self-attention is employed.

The primary purpose of the Category Map Block is to identify parameter sequences and types using cross attention on function signatures while leveraging self-attention for other tokens. This approach ensures that token embeddings are appropriately contextualized based on the nature of the token and the surrounding context. By incorporating category-specific information into token embeddings, the block enhances the model's ability to predict the category and token of the next output in the sequence accurately.

4.3.8 Category Map Decoder

The Category Map Decoder consists of multiple layers, each designed to predict the category of the next token in the sequence. At each layer, the decoder utilizes a common block followed by a Category Map Block to process token embeddings and incorporate category-specific information. The decoder employs a total of 8 attention heads and 6 layers to capture and propagate contextual information effectively. The Category Map Decoder begins by obtaining token embeddings using the Embeddings Manager. These embeddings serve as the input to the decoder layers. Before passing through the decoder layers, dropout regularization is applied to prevent over-fitting and enhance model generalization.

The decoder layers are implemented using an `nn.ModuleList`, allowing for efficient processing and propagation of token embeddings. At each layer, token embeddings undergo processing through the common block and the Category Map Block, enabling the model to capture and leverage category-specific information effectively. At the final layer of the Category Map Decoder, the output is an embedding vector representing the category of the next token in the sequence. This embedding, denoted as E1 (Embedding for Category), encapsulates contextual information and category-specific cues, facilitating accurate prediction of the next token's category.

4.3.9 Category Map Classification Head

The Category Map Classification Head is designed to process token embeddings and predict the category of the next token. The head begins with a feed-forward network, followed by dropout regularization and layer normalization. Subsequently, the output layer projects the hidden dimension into a vocabulary size, enabling the model to generate specific word probabilities. The primary function of the Category Map Classification Head is to predict the category of the next token based on the output of the Category Map Decoder. By applying feed-forward layers, dropout regularization, and layer normalization, the

head enhances the model’s ability to capture and leverage category-specific cues effectively.

The output layer of the Classification Head generates logits, representing the probabilities of each token category. These logits are then processed to determine the most probable category for the next token, facilitating accurate prediction within the decoding process. As Embedding for Category(E1) pass through the head, category probabilities are computed, providing crucial information for the model to generate contextually relevant token sequences.

4.3.10 Category Router

The Category Router is a critical component of the generative Curious Learner model, responsible for routing each token to its specific output token classification head based on its category. It consists of multiple output token classification heads, each corresponding to a unique category present in the training data. These heads are responsible for classifying tokens based on their category probabilities, facilitating accurate routing within the model. The router employs two distinct methods for guiding tokens to their specific routes:

1. **The Hub Method:** The Hub Method involves passing every token to every output token classification head,
2. **The Switch Method:** while the Switch Method routes each token to its specific classification head.

Each method offers unique advantages and trade-offs in terms of training convergence and computational efficiency. The Hub Method offers a straightforward approach by passing every token to every classification head, optimizing training convergence but resulting in redundant calculations. In contrast, the Switch Method optimizes computational efficiency by routing each token directly to its specific classification head, but may encounter challenges with backpropagation during inference due to sequence disruption. Figure 4.4 presents a comprehensive depiction of the architectural framework and logical flow of the category router.

4.3.11 Output Token Block

The Output Token Block is responsible for token generation during the decoding phase. Similar to the Category Map Block, the Output Token Block comprises multiple layers, each consisting of several sub-components. However, unlike the Category Map Block, the Output Token Block does not incorporate cross-attention mechanisms. Instead, all tokens pass through self-attention layers within each block. The primary function of the Output Token Block is to process token embeddings and contextualize them using self-attention mechanisms. By leveraging self-attention, the block enables tokens to capture contextual information from their surrounding tokens, facilitating accurate token generation and sequence completion.

4.3.12 Output Token Decoder

The Output Token Decoder consists of multiple layers, each comprising a series of sub-components. Similar to other decoding blocks within the model, such as

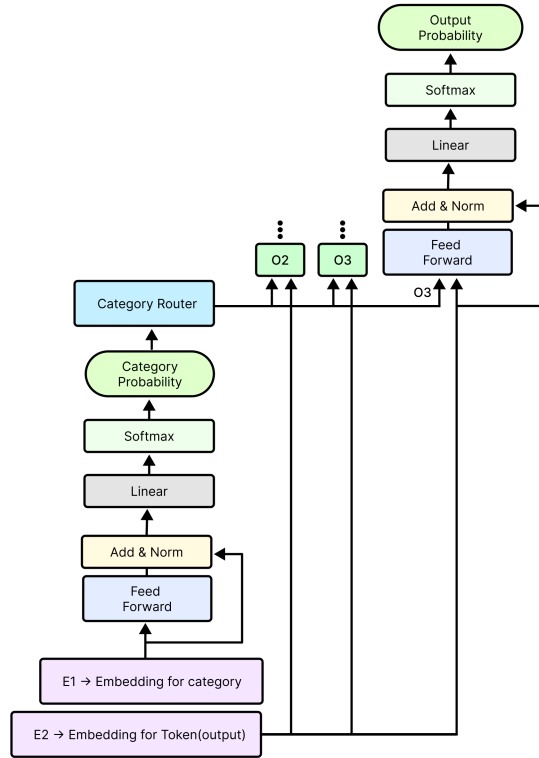


Figure 4.4: Category Router Detailed Architecture Diagram

the Category Map Decoder, the Output Token Decoder utilizes a common block followed by an Output Token Block on each layer. However, unlike the Category Map Decoder, the Output Token Decoder focuses solely on processing output tokens and does not incorporate cross-attention mechanisms.

The primary function of the Output Token Decoder is to decode token embeddings and generate subsequent output tokens based on the contextual information captured from the input tokens. At the final layer of the Output token Decoder, the output is an embedding vector representing the output of the next token in the sequence. This embedding, denoted as E2 (Embedding for Output Token), encapsulates contextual information and output token-specific cues.

4.3.13 Output Token Classification Head

In our model we have multiple output classification heads, each for specific unique category and subcategory combination, which we are calling `OutputTokenClassificationHeadVocabItem`. The category router is responsible for routing each Embedding for output token(E2) through its specific output classification head. These heads are responsible for predicting the probability distribution over the vocabulary of output tokens, enabling the model to generate coherent and contextually relevant token sequences.

The primary function of the Output Token Classification Head is to classify output token embeddings and predict the probability distribution over the vocabulary of output tokens. Each classification head is trained to recognize specific patterns

and generate appropriate output tokens of its data type based on the contextual information encoded in the Embedding for output token(E2).

4.3.14 Response Parser

The primary function of the Response Parser is to process the predicted category probabilities and output token probabilities and generate a human-understandable response. It combines the category probability information with the output token probability distribution to execute functions and parameters and return the result in plain text format. It is the reverse process of the Input Parser(IP). Consider the following example input generated by the Input Parser:

```
[
  (
    "<function MathFunctions.division at 0x117b828c0>",
    { "type":"function", "subType":"float", "subSubType":"execute" }
  ),
  (
    4.5,
    { "type":"float", "subType":"default", "subSubType":"param_one" }
  ),
  (
    2,
    { "type":"integer", "subType":"default", "subSubType":"param_last" },
  )
]
```

After processing by the Response Parser, the generated response would be: "2.25" This response represents the result of executing the division function with parameters 4.5 and 2, yielding the quotient 2.25.

4.4 Architecture Justification

Let's ask some questions to justify our network architecture choice:

1. **What's the difference between Hub and Switch category routing, and why Switch category routing should work better?**

In hub category routing we route each token to all available heads, to keep parallel training intact, so heads that don't own the token will predict nOtMyToKeN. Whereas in switch category routing we route each token to only its specific classification head. So this routing mechanism can remove both unnecessary computation and nOtMyToKeN error with the imbalance token count issue. See Figure 4.5 for details.

The switch method should outperform the hub method because it eliminates the need for nOtMyToKeN tokens in the output token classification heads. This resolves errors of producing nOtMyToKeN from heads meant to generate tokens of specific categories. With only the relevant heads producing tokens, unnecessary computations are reduced, resulting in faster training and inference times.

2. **Why "notmytoken" is introduced in the Hub category routing method?**

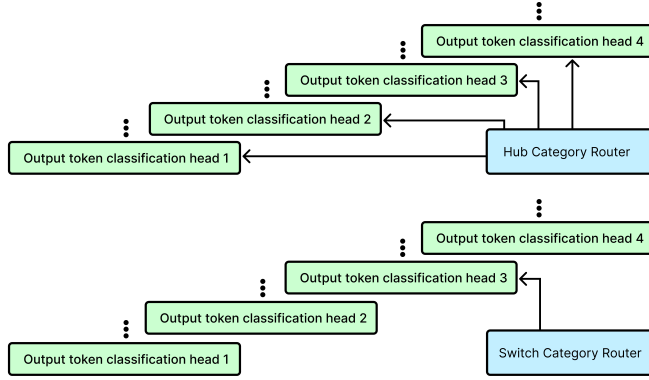


Figure 4.5: Hub vs Switch category routing

The addition of "nOtMyToKeN" serves a purpose in hub category routing by ensuring that all heads produce an output, aligning with machine learning optimization for parallelization. This waste computation resource still it's necessary, as heads not designated for token classification would otherwise lack output without it.

3. Challenges for implementing the Switch category routing method?

Implementing the switch method poses challenges because it disrupts the parallelism of training by guiding tokens exclusively to their relevant classification heads. While we've managed this for inference, ensuring backpropagation during training is complex. Rearranging token order interferes with attention layer backpropagation, and we haven't found a solution yet. Overcoming this obstacle could significantly enhance model accuracy.

4. Can we create the category for function params using function signature instead of predicting them using cross attention?

Although not yet implemented, creating categories based on function signatures rather than predicting function parameter categories can eliminate function parameter category mismatch errors entirely.

4.5 Building Vocabulary

We built and maintained a comprehensive vocabulary for effective tokenization, classification, and decoding. The vocabulary serves as a repository of all unique tokens, categories, and classification heads encountered during training and inference. Vocabulary plays a crucial role in tokenization by providing a structured mapping between raw input data and tokenized representations. During tokenization, input text is parsed and converted into tokens based on predefined hashable vocabulary items.

Moreover, the vocabulary facilitates classification by organizing tokens into distinct classification heads. Each token is associated with a specific category map, which defines its type, subtype, and subtype. Additionally, output tokens are assigned to output token classification heads, which represent the predicted types of tokens generated by the model. By maintaining a comprehensive vocabulary, the Curious

Learner model can effectively tokenize input data, classify tokens into meaningful categories, and generate accurate output predictions.

4.5.1 Category Map Vocabulary Builder

The Category Map Vocabulary Builder module is responsible for managing the vocabulary related to category maps, which represent the type and subtype and sub-subtype of each token. This vocabulary maintains four types of items, each with its own encode and decode functions:

- Index
- CategoryVocabItem: Represents the hashable category map for a token.
- OutputTokenClassificationHeadVocabItem: Represents the hashable type for the output token classification head of the token.
- Input Parser Output

The Category Map Vocabulary Builder dynamically tracks unique category vocab item and output token classification head vocab items. Where each OutputTokenClassificationHeadVocabItem corresponds to an output token classification head. Category vocab item is build using unique category type, subtype and subsubtype combination while output token classification head is build using only the unique category type and subtype combination.

4.5.2 Output Token Vocabulary Builder

The Output Token Vocabulary Builder module manages the vocabulary associated with output tokens, which are the predicted tokens generated by the model. The Output Token Vocabulary Builder is essential for tracking and organizing the various types of output tokens generated by the model. It ensures that the vocabulary is comprehensive and facilitates effective decoding and response generation. This vocabulary maintains three types of items, each with encode and decode functions:

- Index
- OutputVocabItem
- Input Parser Output

4.6 Different Types of Embeddings

In the generative Curious Learner model, embeddings play a crucial role in representing tokens, capturing positional information, and encoding category and task details. Different types of embeddings are utilized to ensure that tokens are accurately represented and contextualized within the model's architecture.

4.6.1 Token Embeddings

Token embeddings serve as the foundational representation of individual tokens within the model. These embeddings capture semantic information about each token, allowing the model to understand and process input data effectively. We have implemented two approaches for generating token embeddings, which encode tokens into dense vector representations based on their contextual usage. The first approach involves using pre-trained transformer models "all-mpnet-base-v2," also known as sentence encoders. The second approach utilizes our custom tokenizer, which we built from scratch using vocabulary builder and the nn.embeddings layer of the PyTorch library. In our experiments, we have observed that our custom tokenizer yields better results. For function tokens we used the doc string of the function to generate the embeddings using the "all-mpnet-base-v2," pre-trained sentence encoder model. But for cross attention in the category map block we used the signature of the function and "microsoft/graphcodebert-base" pre-trained model to get the embeddings.

4.6.2 ALIBIBI Embeddings

ALIBIBI embeddings incorporate positional information into token representations, enabling the model to capture long-range dependencies and context information. These embeddings are generated using the ALIBIBI encoder, and subtracted from the calculated attention score in the Multi head attention to introduce positional biases. By incorporating positional information, ALIBIBI embeddings enhance the model's ability to understand the sequential structure of input data and make accurate predictions. Token in question will have zero bias while furthest token will have most bias. For adding randomness between the heads we gradually decrease the base by a factor of $\frac{1}{2}$ in each upcoming head from the previous. Please refer to Figure 4.6 for details.

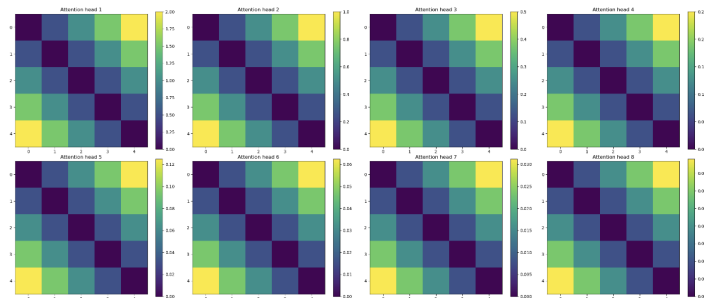


Figure 4.6: ALiBiBi Bias for All Attention Heads

4.6.3 Category and Task Embeddings

Category and task embeddings encode additional metadata about tokens, including their category type, subtype, and task information. These embeddings are generated using the category and task encoder module, which adds category and task information to token embeddings. By incorporating category and task details, combining these embeddings with token embeddings enable the model to differentiate between different types of tokens and perform task-specific operations more effectively.

4.6.4 Combined Embeddings

Combined embeddings merge token embeddings, and category/task embeddings into a unified representation for each token. These embeddings capture comprehensive information about tokens, including their semantic meaning and category/task details. Combined embeddings are used as input features for the model’s training and inference processes, providing a rich representation of input data that facilitates accurate prediction and generation tasks.

4.7 Saving, Loading, and Retraining the Model

The Curious Learner model employs a checkpoint manager to save and load its state during training and inference. This functionality is crucial for preserving the model’s progress and parameters, allowing for seamless continuation of training or retrieval of trained models for inference purposes.

When saving the model, the checkpoint manager stores various components of the model’s state, including the following key information:

- Epoch: The current epoch number of training.
- CL Model State: The state of the CL model, containing its parameters and optimizer state.
- Optimizer State: The state of the optimizer used for training.
- Category Map Decoder State: The state of the category map decoder module.
- Category Map Classification Head State: The state of the category map classification head module.
- Output Token Decoder State: The state of the output token decoder module.
- Category Router State: The state of the category router module.
- Category Map Decoder Blocks State: The state of the blocks within the category map decoder.
- Output Token Decoder Blocks State: The state of the blocks within the output token decoder.
- Output Token Classification Heads State: The state of the dynamically created output token classification heads.
- Embeddings Layer State: The state of the embeddings layer, which is used for generating the initial token embeddings.

Upon loading the model, it is essential to use the same input examples as those used during training to ensure proper reconstruction of the output token classification heads. This is required because we used dynamic calculation of the output token classification heads from the input data. This ensures that the model’s architecture remains consistent and avoids tensor size mismatches. However, once the model is loaded, it can be retrained on new input data as long as the input tokens are

recognized by the model. The ability to save, load, and retrain the model provides flexibility and scalability, enabling efficient management and utilization of the Curious Learner across various tasks and scenarios.

4.8 Data Loader

The Data Loader package plays a crucial role in preparing the data for training and testing of the CL-Pre-Trainer model. It comprises two main components: the Data Generator and the Batch Builder.

4.8.1 Data Generator

The Data Generator module is responsible for generating samples upon request. It provides a mechanism to create data samples for training and testing purposes. It supports both random data loading and also loading data using identifier. These samples are then processed by the Data Loader to incorporate necessary information for model training. As an illustration, consider the following example:

```
[
  {
    "inputStr": "##multiplication(107.23600916441234,784.625535184504)",
    "outputStr": "107.23600916441234 times 784.625535184504 equals?",
    "taskType": "func_to_nl_translation",
    "sentence": "##multiplication(107.23600916441234,784.625535184504)
                = 107.23600916441234 times 784.625535184504 equals?",
    "ioParserOutput": [
      {
        "token": "<BOS>",
        "category": { "type": "special", "subType": "word", "subSubType": "none" },
        "position": 0
      },
      ...
    ],
    "inputTokenCount": 4,
  }
]
```

In this example, we have a data sample for the task of function-to-natural-language translation. The input string represents a multiplication operation, and the output string prompts the user to calculate the result. The input is tokenized and processed into a sequence of tokens with corresponding category information, including special tokens such as <BOS>(beginning of sequence). The "inputTokenCount" field indicates the number of tokens in the input sequence.

4.8.2 Batch Builder

The Batch Builder module enhances the data generated by the Data Generator by adding essential information required for training the CL-Pre-Trainer model. This includes details such as input and output strings, task types, and pre-processed tokenized representations of the input and output sequences. Additionally, it facilitates the construction of batches with specified batch sizes and maximum decoder sequence lengths.

The batch builder function, performs various tasks such as adding special tokens like <BOS>(beginning of sequence) and <EOS>(end of sequence), truncating sequences

to meet length constraints, and preparing attention masks for model training. Furthermore, it supports generative training by employing a <MASK>token to indicate the next token to predict, facilitating sequential prediction until the <EOS>token is reached.

Together, the Data Generator and Batch Builder modules form an integral part of the data loading pipeline for the CL model, ensuring that the model receives properly formatted and processed data for effective training and evaluation. Consider the following example of a batch:

```
[
  {
    "inputStr": [
      {
        "token": "<BOS>",
        "category": { "type": "special", "subType": "word", "subSubType": "none" },
        "position": 0
      },
      ...
      {
        "token": "<MASK>",
        "category": { "type": "special", "subType": "word", "subSubType": "none" },
        "position": 7
      },
      ...
    ],
    "outputStr": [
      {
        "token": "<BOS>",
        "category": { "type": "special", "subType": "word", "subSubType": "none" },
        "position": 0
      },
      ...
      {
        "token": <function MathFunctions.subtraction at 0x1241d5ab0>,
        "category": { "type": "function", "subType": "integer", "subSubType": "execute" },
        "position": 7
      },
      {
        "token": "<MASK>",
        "category": { "type": "special", "subType": "word", "subSubType": "none" },
        "position": 8
      },
      ...
    ],
    "taskTypeKey": "nl_to_func_translation",
    "initialTokenCountKey": 7
  },
  ...
]
```

In this example, we have a batch of data samples for the task of natural language to function translation. Each sample contains input and output sequences represented as lists of tokens, along with their corresponding categories and positions. The "taskTypeKey" field indicates the type of task associated with the batch, and the "initialTokenCountKey" field denotes how many tokens need to be given to the model for first next word prediction.

4.9 Training and Inference Methods

The training and inference methods for the CL model encompass various approaches tailored to different scenarios and objectives. These methods involve guiding tokens through specific routes to output token classification heads, which play a crucial role in predicting the next token during training and inference.

4.9.1 Guiding Tokens to Output Token Classification Heads

Two main methods are employed to guide tokens to their specific routes:

1. Hub Method: In the hub method, every token is passed to every output token classification head. If the token does not match the classification head, it is identified as 'notMyToken'. This approach allows for efficient utilization of the attention layer, leading to faster training convergence. However, it involves redundant calculations.
2. Switch Method: In the switch method, each token is passed only to its specific output token classification head. While this approach is more optimized, it may lead to longer backpropagation times due to the disruption of sequential word order in the attention layer. Therefore, it is primarily used during inference. Note that due to time constraint and parallelization issue we can't implement the switch training method properly.

4.9.2 Training and Inference Types

Training and inference can occur in four different modes, depending on whether it is generative or non-generative and whether the hub or switch method is employed:

1. Generative Hub Training/Inference: This mode entails training or inferring the model in a generative manner using the hub method.
2. Generative Switch Training/Inference: Similar to generative hub training or inference, but employs the switch method to guide tokens during training or inference.
3. Non-Generative Hub Training/Inference: In this mode, the model undergoes training or inference in a non-generative manner using the hub method.
4. Non-Generative Switch Training/Inference: Similar to non-generative hub training or inference, but utilizes the switch method to guide tokens.

4.9.3 Training Process

The Curious Learner model undergoes two stages of training: non-generative and generative. Additionally, training using the Hub Method requires inference using the Hub Method, and likewise for the Switch Method.

Non-generative Training:

Initially, the model learns to predict the next word in an autoregressive manner without using any <MASK>token. Here, we use all the tokens with future masks and predict the right-shifted result of the tokens in a single pass. Like: BOS I want to eat an apple >> I want to eat an apple EOS. This process employs teacher forcing and parallel training to expedite the fixation of the embedding layer and comprehension of natural language. It serves as the foundational step for subsequent training.

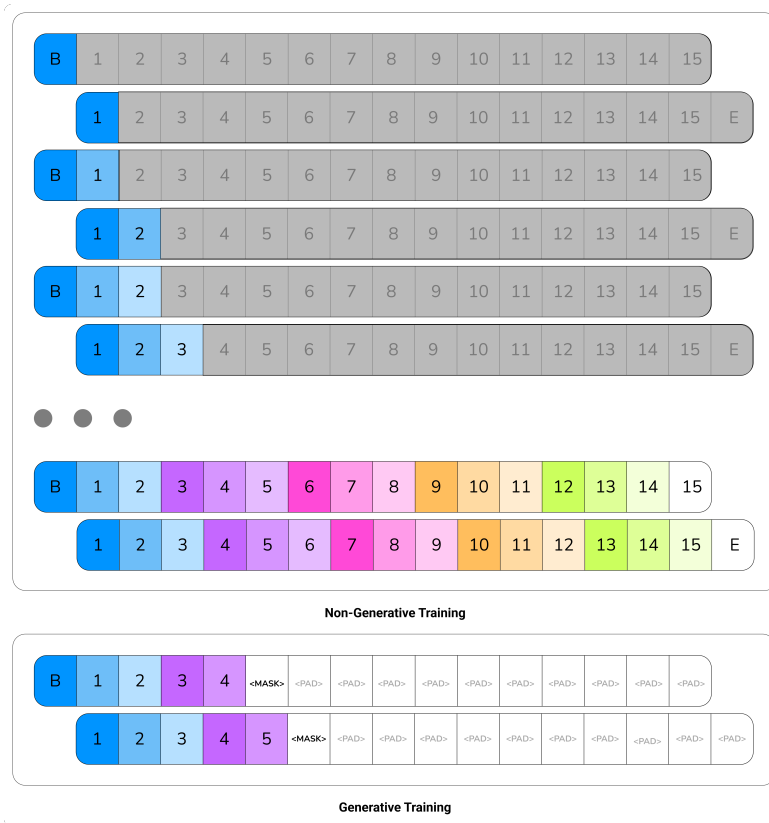


Figure 4.7: Non-generative and Generative Training

Generative Training:

After completing non-generative training, the model is further trained using a generative approach. Here in a single pass, we just predict the `<MASK>` token and shift the `<MASK>` to the next position with all the following tokens as padding. We run generative training in a recursive loop both in training and inference until EOS is reached, the only difference is in training we use teacher forcing while in inference we don't. This helps the model to become proficient as a generative next token predictor where in each step the model predicts the next word. Please refer to Figure 4.7 for details.

XLNet[19] offers an advanced autoregressive pre-training approach, addressing limitations of BERT[14] by effectively modeling bidirectional contexts. It maximizes the expected likelihood over all permutations of the factorization order, benefiting from its autoregressive training nature. Both the training approach of XLNet[19] and BERT[14] can significantly improve training results, albeit at the cost of substantial time and computational resources.

Chapter 5

Result

5.1 Hyperparameter Selection

In this section, we discuss the various hyperparameters that were experimented with and the choices we made based on the performance of the CL model.

5.1.1 Activation Function

Activation functions play a crucial role in neural networks by introducing non-linearity, allowing the model to learn complex patterns in the data. We experimented with several activation functions, including Swish, PReLU, and ReLU.

Swish:

Swish is a recently proposed activation function that has been shown to outperform traditional activations like ReLU in some cases[25].

It is defined as $f(x) = x \cdot \text{sigmoid}(x)$.

SwiGLU:

SwiGLU, short for Swish-Gated Linear Unit, is a type of activation function that combines elements of the Swish activation function with a gating mechanism. SwiGLU aims to capture both the non-linearity of the Swish function and the linearity of the linear unit, providing a flexible activation function for neural networks[34].

It is defined as: $f(x) = \sigma(x) \cdot x + (1 - \sigma(x)) \cdot x$

PReLU (Parametric Rectified Linear Unit):

PReLU introduces learnable parameters to Leaky ReLU, enabling the model to adaptively learn the optimal slope of the negative part of the activation function[16].

ReLU (Rectified Linear Unit):

ReLU is a simple and widely used activation function that outputs zero for negative inputs and the input value for positive inputs[4].

After experimentation, we found that PReLU performed best for our use case, providing improved convergence and performance.

5.1.2 Normalization

Normalization techniques help stabilize and speed up the training process by ensuring that input features are on a similar scale. We experimented with Layer Normalization and RMSNorm.

Layer Normalization:

Layer Normalization normalizes the activations of each layer independently, across the features dimension.

RMSNorm (Root Mean Square Normalization):

RMSNorm normalizes the activations using the root mean square of the activations over the entire feature dimension[20].

We found that RMSNorm worked well for our model, contributing to faster convergence and improved performance.

5.1.3 Regularization

Regularization techniques are employed to prevent over-fitting and improve the generalization ability of neural networks. One common regularization method is dropout[5], which randomly drops a fraction of the input units during training. This helps prevent complex co-adaptations between neurons and reduces over-fitting. We employed nn.Dropout to apply dropout with a dropout probability of 0.01.

5.1.4 Learning Rate Scheduler

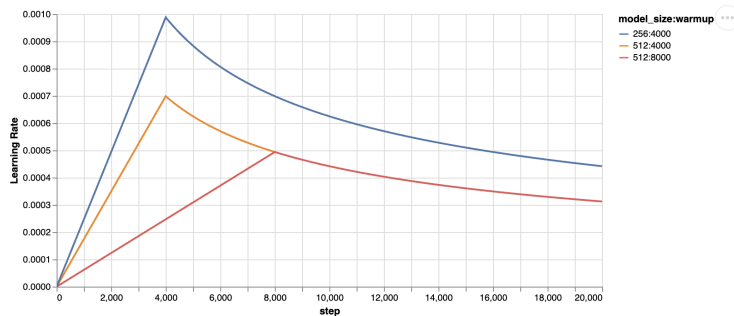


Figure 5.1: Learning Rate Scheduler Graph[15]

The learning rate scheduler[15] dynamically adjusts the learning rate during training to facilitate effective optimization. We employed the Adam optimizer with a learning rate scheduler, which gradually increases the learning rate before decaying it [Figure: 5.1].

We utilized three parameters to control the learning rate scheduler:

- Factor: This parameter determines the rate at which the learning rate decreases. A factor of 0.05 indicates that the learning rate is multiplied by this factor after each step.

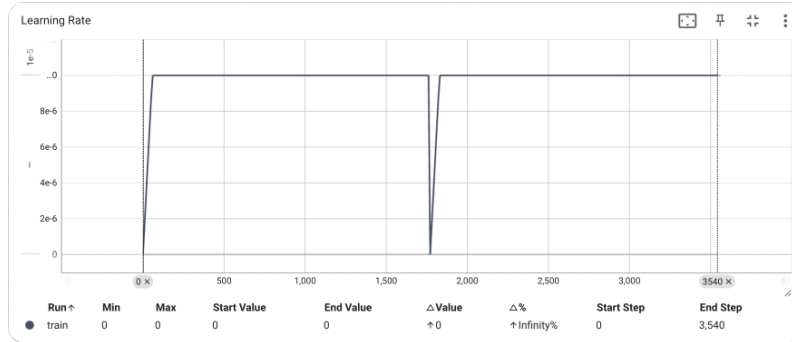


Figure 5.2: Curious Learner Learning Rate Scheduler Graph

- Warmup: Specifies the number of steps during which the learning rate is increased. We used $(\text{number of batch} * \text{epoch}) // 2$ for calculating the warmup number.
- Max Rate: Sets an upper limit on the learning rate to prevent it from growing too large.

5.1.5 Optimizer

We used the AdamW[11] optimizer, a variant of the Adam optimizer that includes weight decay regularization. Weight decay adds a penalty term to the loss function, encouraging the model to learn smaller weight values and reducing over-fitting.

5.1.6 Criterion

For the criterion, we employed the CrossEntropyLoss[2] with label smoothing[17]. Label smoothing[17] is a regularization technique that prevents the model from becoming overconfident in its predictions by smoothing the target distribution.

Additionally, we applied weighted loss[29] to address class imbalance issues. Weights were calculated based on the frequency of each class in the dataset, with a higher weight assigned to underrepresented classes. We used 0.3 weight for the nOtMyToKeN token irrespective of the calculated value.

As we have multiple head for output token classification, we used the multitask learning approach here. In this approach, the network is trained to perform multiple tasks simultaneously by optimizing a combined loss[10] function that aggregates the losses from each task. In our case from each output token classification head. This can be beneficial as the classification tasks are related and learning one classification can help improve performance on another.

5.1.7 Epoch

The training process of the CL model spanned multiple epochs, with adjustments made to the training approach and hyperparameters over time.

Non-Generative Training (40 Epochs):

Initially, the model was trained for 40 epochs in a non-generative manner. During this phase, the focus was on learning the underlying structure of natural language without considering the generative aspect.

Generative Training:

Following the non-generative training, the trained model was loaded, and generative training commenced. This phase lasted until convergence, during which the model was trained to predict the next token in a sequence, generating natural language text. To improve the accuracy of the generated sequences, the training process was adjusted. The training process involved dynamic adjustments to the batch size. Initially, a larger batch size was used, and it was gradually decreased to 4 over the course of training. Additionally, after 20 epochs of generative training, the model's performance was evaluated, and the best-performing model based on accuracy was saved. This approach allowed for the preservation of the best model's state for further training or evaluation.

5.1.8 Training Batch Size

The choice of batch size is a crucial hyperparameter in training deep learning models, as it affects both computational efficiency and the quality of the learned representations. In the case of the CL model, the training batch size underwent dynamic adjustments during the training process. The training commenced with a relatively large batch size to leverage parallel processing and accelerate the training process. A larger batch size often leads to faster convergence but may compromise the model's ability to generalize. As training progressed, the batch size was gradually reduced to a smaller value. This reduction allowed for finer updates to the model parameters and improved convergence to more optimal solutions. Eventually, the batch size was decreased to a smaller value, specifically set to 4. This smaller batch size enabled the model to capture finer patterns in the data and potentially achieve better generalization performance.

5.1.9 Number of Heads in Attention Layer

We employed 8 heads in the attention layer, consistent with the architecture outlined in the "Attention is All You Need"[12] paper.

5.1.10 Number of Layers in Decoder

Similar to the referenced paper, our decoder consisted of 6 layers to facilitate learning complex patterns.

5.1.11 Hidden Embeddings Dimension

The hidden embeddings dimension was set to 768 for our model, as it proved effective in capturing intricate relationships within the data.

5.1.12 Feed Forward Layer Dimension

We opted for a feed forward layer dimension of 2048, a choice that allowed the model to capture intricate feature representations effectively. This dimensionality was applied consistently across different components, including the category map block, output token block, and classification heads. In the category map block & output token block, the feed forward layer was designed to transform the input embeddings using two linear transformations with a Leaky ReLU activation function in between. This architecture facilitates the extraction of meaningful features from the token embeddings. Similarly, within the classification heads, the feed forward layer was constructed to process the token embeddings, employing two linear transformations with a Parametric Rectified Linear Unit (PReLU) activation function.

5.1.13 Max Decoding Length

A max decoding length of 24 was chosen to balance computational efficiency and model performance.

5.1.14 Add BOS and EOS Tokens

Both Beginning of Sequence (BOS) and End of Sequence (EOS) tokens were added during training and inference to mark the beginning and end of sequences, respectively. This facilitated sequence generation tasks.

5.1.15 Data Loader Parameters

The data loader function is responsible for loading and mixing samples from different tasks to create training data for the Curious Learner model. We have 4 tasks, 98 generators(functions), and for each unique task, generator pair we have 20 to 30 samples. It accepts various parameters to customize the loading process:

- `seed`: Seed for generating random indexes.
- `shuffle`: Flag to shuffle the list before returning if set to true.
- `batch_size`: Size of the batch.
- `number_of_batch`: The number of batches needed, where $batch_size \times number_of_batch$ equals the total count of samples.
- `add_bos_and_eos`: Flag indicating whether to add the Beginning of Sentence (BOS) and End of Sentence (EOS) tokens to the token list.
- `max_sequence_length`: Maximum length until which padding will be added. If set to *None*, no padding will be applied.
- `task_generator_indexes`: Index indicating the task-type generator, ranging from 0 to 3.
- `generator_indexes`: Index of the example function generator, ranging from 0 to the length of the sample generators list.

- identifier: Example function identifier. Samples retrieved start from this identifier and continue till the identifier + `batch_size`.
- param_variation: We can generate an infinite number of samples by introducing variations to the parameter. This allows us to control the sample creation process.

The `task_generator_indexes` parameter corresponds to four available task types, each associated with a specific set of generators. These generators include natural language to natural language (N12N1Samples), function to function (F2FSamples), function to natural language (F2NSamples), and natural language to function (N2FSamples) translation. Each task type comprises multiple generators, and each generator contains a set of samples. `generator_indexes` is a list which, let you select which generator(functions) to use for generating samples. The identifier and `batch_size` parameters work together to select samples from the specified generators. Samples are chosen starting from the identifier and extending up to identifier + `batch_size`.

5.2 Accuracy Increment per Epoch

The accuracy increment per epoch provides insight into the model’s performance improvement over successive training epochs. By monitoring the accuracy metric across epochs, we can observe how effectively the model learns and generalizes from the training data.

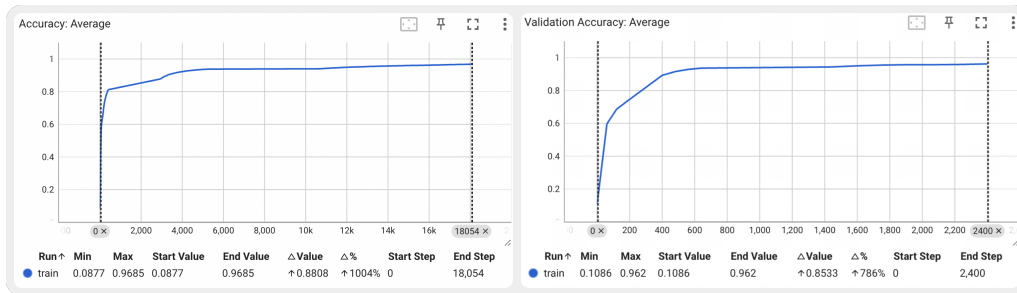


Figure 5.3: Average Accuracy vs Epoch Graph

During training, we tracked the accuracy of the model on a validation set at the end of each epoch. The accuracy increment per epoch represents the change in accuracy from one epoch to the next, indicating whether the model is making progress or plateauing in its learning. A consistent increase in accuracy across epochs suggests that the model is learning relevant patterns and improving its predictive performance as shown in Figure 5.3.

In our experiments, we observed a steady increase in accuracy over the training epochs, indicating that the model was effectively learning from the training data and improving its performance over time. We find the accuracy of the category map classification head and most of the output token classification heads become 1 after some time, except the word, floating point, and integer output token classification head as shown in Figure 5.4. As some data are continuous in nature like integer or floating point so we think using a custom data type-specific tokenizer for these output token heads can solve this issue easily.

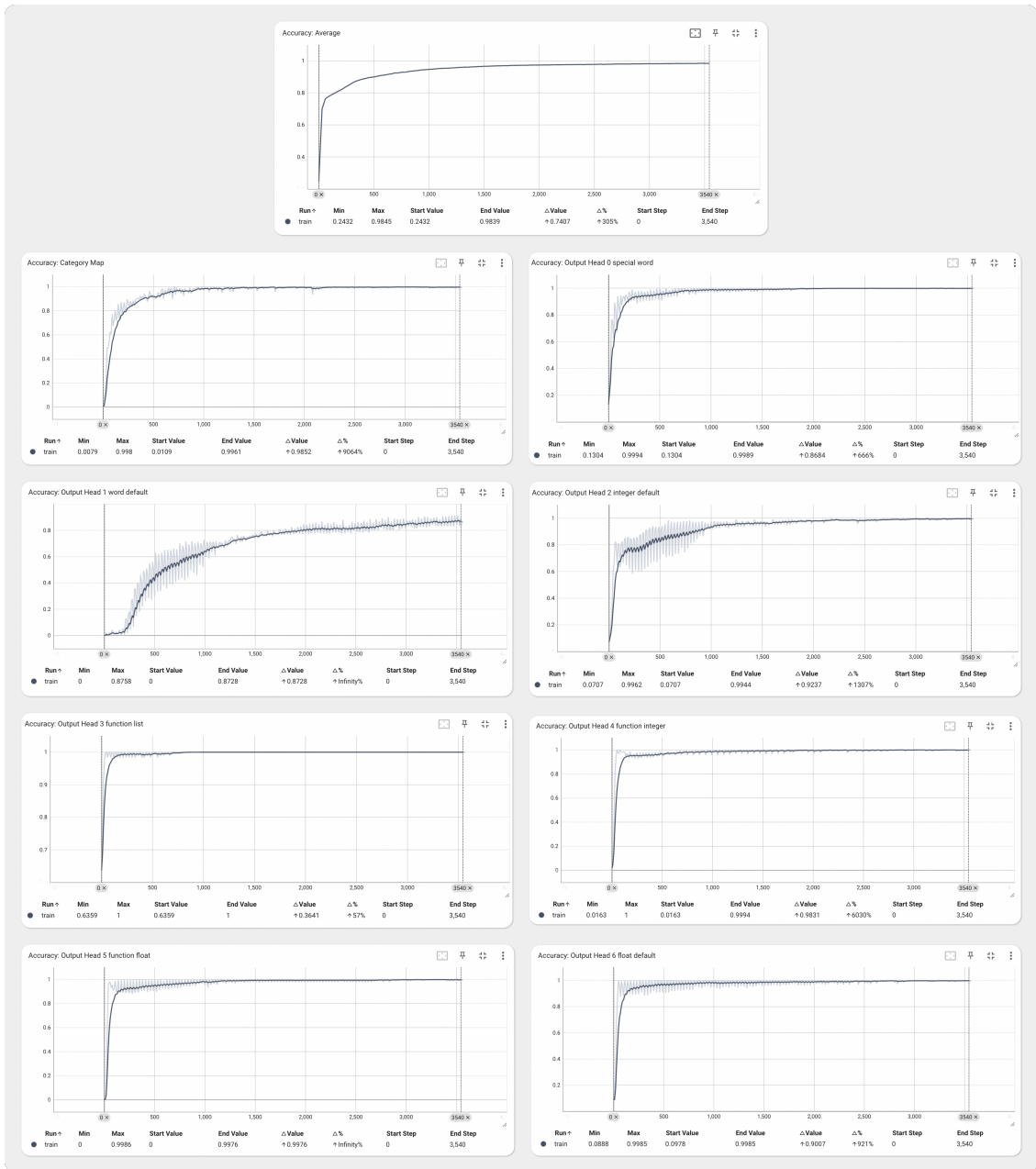


Figure 5.4: Accuracy vs Epoch Graph

5.3 Loss Decrement per Epoch

The loss decrement per epoch provides insight into how the model’s loss function decreases over successive training epochs. Monitoring the loss decrement helps assess the model’s convergence and optimization progress during training.

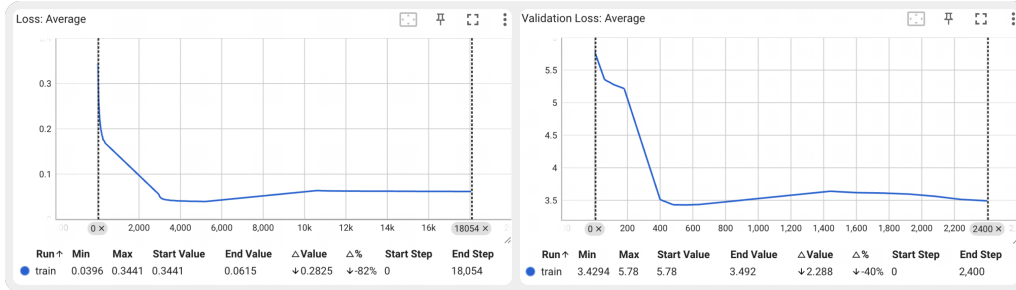


Figure 5.5: Average Accuracy vs Epoch Graph

During training, we also tracked the loss of the model on a validation set at the end of each epoch. The loss decrement per epoch represents the change in loss from one epoch to the next, indicating whether the model is converging towards an optimal solution or plateauing in its optimization. A consistent decrease in loss across epochs suggests that the model is effectively minimizing its error and learning relevant patterns from the training data as shown in Figure 5.5.

In our experiments, we observed a consistent decrease in loss over the training epochs, indicating that the model was effectively minimizing its error and learning relevant patterns from the training data. We find that the loss of the category map classification head and most of the output token classification heads steadily decreased over time, converging towards lower values Figure 5.6. However, some heads, such as those associated with word, floating point, and integer output tokens, exhibited slower rates of loss reduction. We attribute this observation to the inherent challenges in distinguishing between similar initial embeddings, especially when utilizing a sentence encoder for obtaining initial embeddings rather than a more specialized tokenizer. Addressing this issue may require the implementation of a custom data type-specific tokenizer tailored to these output token heads, potentially improving the model’s performance and convergence.

5.4 Inference

Given limitations in both data availability and computational power, achieving universality, validity, or general applicability in our model, which contains 111 million trainable parameters, is unattainable. Consequently, accuracy serves as a primary evaluation metric, indicating the extent to which our model memorizes correctly. On average, for a batch comprising 97 samples, we attain the scores outlined in Table 5.1. For additional insights, please refer to Figure 5.8.

Initially, we trained our model to determine if it could memorize samples. We choose our 4 tasks, 50 functions, and 4 samples each, to create a dataset totaling 800 samples. After several hours of training on a T4 GPU, we achieved the following average performance metrics across a batch of 98 samples: Accuracy of 0.85, BLEU[1] Score of 0.84, and Perplexity[6] Score of 5.9. Afterward, we created a dataset with 4 tasks,

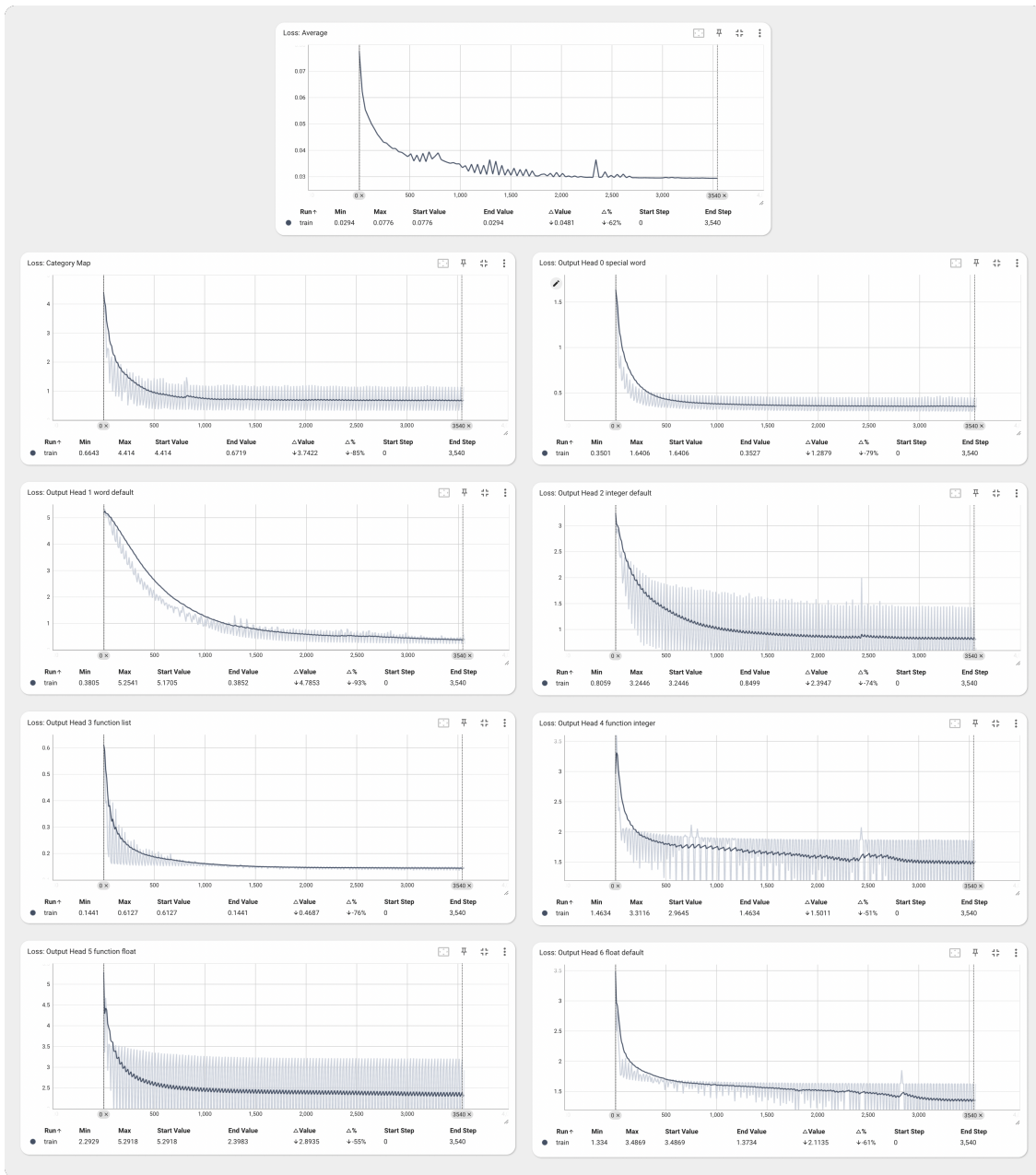


Figure 5.6: Loss vs Epoch Graph

Metric	Value
Accuracy	0.85
BLEU Score	0.84
Perplexity	5.9

Table 5.1: Performance Metrics

10 functions, 4 samples, and 100 parameter variations each, totaling 16,000 samples. However, due to computational limitations, we couldn't process this dataset. So, we downscaled to 400 samples with 4 tasks, 10 functions, 1 sample, and 10 parameter variations each. We split these samples into 80% training, 10% validation, and 10% testing sets.



Figure 5.7: Parsed Result Example for Each Task

After training our model for approximately 150 epochs, 40 non-generative and the rest generative, we achieved an accuracy of 0.82, BLEU score of 0.82, and Perplexity score of 5.15 on the testing set. While these results look promising, they are likely memorized due to the small dataset size. In some cases, our model overfit and predicted memorized parameters from the training data instead of generalizing. For example: "When asked how many unpacked apples are left if 20 apples are packed into boxes of 10 each, the model incorrectly predicted 42 and 32 for the modulus function instead of the correct values 20 and 10."

While these scores provide valuable insights into overall performance, they do not pinpoint specific areas of error. In order to ascertain the weaknesses of the model, we

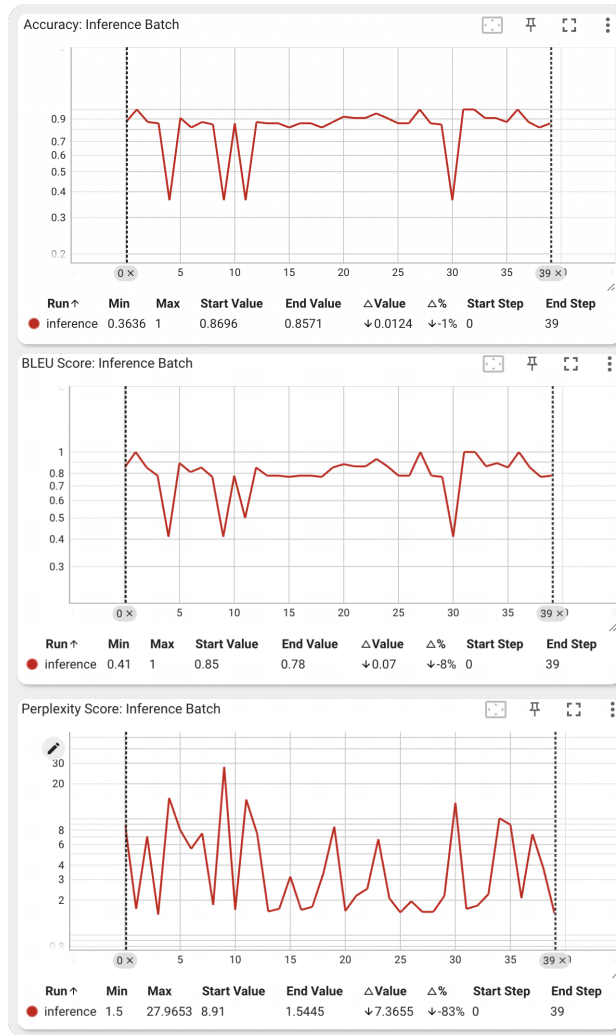


Figure 5.8: Inference Evaluation Metrics vs Epoch Graph

provide examples showcasing both accurately parsed results and inaccurately parsed results for each task (refer to Figure 5.7). Our analysis reveals that a common source of error lies in the model's tendency to predict the 'notMyToken' instead of the correct token, leading to parsing inaccuracies. We posit that training and inference using the switch method, without the 'notMyToken' for each output token classification head, could mitigate this issue.

Besides, as we are training on a tiny dataset our model is memorizing the parameters sometimes, which we believe can be solved if we train for less epoch and use larger sample datasets.

Chapter 6

Discussion

6.1 Challenges and Limitations

Although the results were promising, we still faced some issues. These are described below:

1. Achieving comprehensive natural language understanding in a model necessitates substantial computational resources, extensive datasets, and significant engineering efforts. However, our current project scope does not accommodate the requirements for training such a highly generalizable model. The limitations in available resources, including both data and computational power, preclude the training of a model that can effectively generalize across diverse linguistic contexts.
2. Function overloading could not be implemented, as our system does not permit the retention of multiple signatures with the same name.
3. We utilized a category router to route tokens to their specific classification head. However, there were two potential methods for doing this. We initially attempted both approaches but, due to time constraints, we were unable to implement the Switch method effectively. Consequently, we utilized the Hub method, which involved routing tokens to each classification head using the 'nOtMyToKeN' approach for tokens that were not intended to be classified by other classification heads.
 - (a) Switch method (Optimized) - This method routed each token to its specific classification head only.
 - (b) Hub method (Non-Optimized) - This method routed each token to each classification head.
4. Initially, we considered implementing a Siamese network to establish relationships between functions. However, this approach quickly became impractical as the model's size increased, leading us to abandon the idea.

6.2 Future Works

In the future, we aim to enhance our model through the following:

1. Employing a tokenizer for each data type, to ensure comprehensive and accurate processing. As continuous data type like integer, floating point involves converting them into discrete tokens that can be processed by models.
2. Improve the switch training method for a more optimized training cycle. Although hub training and switch inference are functional, enhancing the switch training process could streamline both training and inference methods, eliminating redundant computations.
3. Cross-attention mechanisms are unnecessary for discerning function parameters. By predicting the function token first, we leverage the subsequent function signature to discern the required token types. Thus, instead of predicting the next token category map, we construct it using available information and subsequently predict the output token. This approach can enhance both the efficiency and accuracy of our model.

6.3 Conclusion

The Curious Learner model, initially a vanilla Transformer, evolved into a decoder-only generative model trained on four tasks. With a customized vocabulary and sentence encoder, it excelled in function execution via natural language. Its innovative approach to language understanding opens doors to enhanced automation and problem-solving, promising transformative advancements in human-computer interaction and task automation. However, addressing key challenges such as better initial embedding for continuous data, creating function parameter category types using function signatures instead of predicting them, and implementing a parallelizable Switch category routing mechanism could not only decrease our computation need but also lead to drastic improvements in our results.

The code for training and evaluating our models is on GitHub:

<https://github.com/nerdslab-club/cl.model>

Additionally, sample input-output pairs for the identified four tasks are available at

<https://github.com/nerdslab-club/cl.data/tree/cl.model.main>

Moreover, documentation for the selected 98 mathematical functions can be found at:

https://github.com/nerdslab-club/function_representation

Bibliography

- [1] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “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.
- [2] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.
- [3] S. Harnad, “Symbol grounding problem,” *Scholarpedia*, 2007. DOI: 10.4249/SCHOLARPEDIA.2373.
- [4] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [5] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint arXiv:1207.0580*, 2012.
- [6] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent neural network regularization,” *arXiv preprint arXiv:1409.2329*, 2014.
- [7] L. A. Dennis, M. Fisher, M. Slavkovik, and M. Webster, “Formal verification of ethical choices in autonomous systems,” *Robotics and Autonomous Systems*, 2016. DOI: 10.1016/J.ROBOT.2015.11.012.
- [8] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, “Rasa: Open source language understanding and dialogue management,” *arXiv preprint arXiv:1712.05181*, 2017.
- [9] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” in *International conference on machine learning*, PMLR, 2017, pp. 1243–1252.
- [10] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, “Deep gradient compression: Reducing the communication bandwidth for distributed training,” *arXiv preprint arXiv:1712.01887*, 2017.
- [11] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.
- [12] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [13] D. Cer, Y. Yang, S.-y. Kong, *et al.*, “Universal sentence encoder,” *arXiv preprint arXiv:1803.11175*, 2018.

- [14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [15] A. M. Rush, “The annotated transformer,” in *Proceedings of workshop for NLP open source software (NLP-OSS)*, 2018, pp. 52–60.
- [16] L. B. Godfrey, “An evaluation of parametric activation functions for deep learning,” in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, IEEE, 2019, pp. 3006–3011.
- [17] R. Müller, S. Kornblith, and G. E. Hinton, “When does label smoothing help?” *Advances in neural information processing systems*, vol. 32, 2019.
- [18] A. Murali and P. Madhusudan, “Augmenting neural nets with symbolic synthesis: Applications to few-shot learning,” *arXiv: Learning*, 2019.
- [19] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” *Advances in neural information processing systems*, vol. 32, 2019.
- [20] B. Zhang and R. Sennrich, “Root mean square layer normalization,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [21] T. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [22] K. Ellis, C. Wong, M. Nye, *et al.*, “Dreamcoder: Growing generalizable, interpretable knowledge with wake–sleep bayesian program learning,” *Philosophical Transactions of the Royal Society A*, 2020. DOI: 10.1098/RSTA.2022.0050.
- [23] D. Guo, S. Ren, S. Lu, *et al.*, “Graphcodebert: Pre-training code representations with data flow,” *arXiv preprint arXiv:2009.08366*, 2020.
- [24] J. Huang, C. Smith, O. Bastani, R. Singh, A. Albarghouthi, and M. Naik, “Generating programmatic referring expressions via program synthesis,” *International Conference on Machine Learning*, 2020.
- [25] M. A. Mercioni and S. Holban, “P-swish: Activation function with learnable parameters based on swish activation function in deep learning,” in *2020 International Symposium on Electronics and Telecommunications (ISETC)*, IEEE, 2020, pp. 1–4.
- [26] C. Raffel, N. Shazeer, A. Roberts, *et al.*, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.
- [27] J. Austin, A. Odena, M. Nye, *et al.*, “Program synthesis with large language models,” *ArXiv*, 2021.
- [28] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, and Y. Yue, “Neurosymbolic programming,” *Found. Trends Program. Lang.*, 2021. DOI: 10.1561/25000000049.
- [29] K. R. M. Fernando and C. P. Tsokos, “Dynamically weighted balanced loss: Class imbalanced learning and confidence calibration of deep neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 7, pp. 2940–2951, 2021.

- [30] O. Press, N. A. Smith, and M. Lewis, “Train short, test long: Attention with linear biases enables input length extrapolation,” *arXiv preprint arXiv:2108.12409*, 2021.
- [31] OpenAI. “Chatgpt plugins.” (2023), [Online]. Available: <https://openai.com/index/chatgpt-plugins> (visited on 03/23/2023).
- [32] OpenAI. “Function calling.” (2023), [Online]. Available: <https://platform.openai.com/docs/guides/function-calling> (visited on 03/23/2023).
- [33] Y. Shavit, S. Agarwal, M. Brundage, *et al.*, “Practices for governing agentic ai systems,” *Research Paper, OpenAI, December, 2023*.
- [34] D. Groeneveld, I. Beltagy, P. Walsh, *et al.*, “Olmo: Accelerating the science of language models,” *arXiv preprint arXiv:2402.00838*, 2024.
- [35] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, “Roformer: Enhanced transformer with rotary position embedding,” *Neurocomputing*, vol. 568, p. 127 063, 2024.
- [36] Significant Gravitas, *AutoGPT*. [Online]. Available: <https://github.com/Significant-Gravitas/AutoGPT>.