XiaoFish - Natural language executes terminal command system

Jingyuan Chen, YiFei Guo, Shu Li, YanChen Li, ZiRui Liu, WenZhe Ma, and Qingyuan Mao

Syracuse University

Abstract

With the rapid development of computers, the terminal command line interface (CLI) has become the preferred tool for many developers and system administrators due to its efficient resource management and operational flexibility. However, the use of CLI is difficult and cumbersome for most non-technical users, requiring users to spend a lot of time learning and memorizing complex terminal commands. Even expert users find it difficult to remember all commands perfectly. This high threshold not only limits the convenience of use for ordinary users, but also increases the threshold for novices. Therefore, this paper proposes to build a system that executes terminal commands in natural language, allowing users to interact with the terminal through daily language, thereby greatly reducing the threshold for use and improving the user's operating experience. The system in this paper will be built based on the transformer model and trained on the nl2bash dataset. The test method uses the bleu score and a test method we innovatively proposed. Overall, we get nearly about **90%** accuracy by our test method, and also have **81.70%** BLEU score. This result shows our XiaoFish system works excellent in nl2bash dataset.

Keywords: Natural Language Processing, Command Line Interface, Transformer Model, nl2bash, BLEU Score

Contents

1 Introduction

The dream of using natural language (NL) to program computers dates back to the early days of computing itself (Sammet, 1966). While natural language lacks the precision of formal programming languages (Dijkstra, 1978), it is universally accessible and particularly well-suited for automating repetitive tasks such as file operations, searches, and application-specific scripting (Wilensky et al., 1984, 1988; Dahl et al., 1994; Quirk et al., 2015; Desai et al., 2016). In recent years, advancements in natural language processing (NLP) and neural networks have paved the way for systems capable of bridging the gap between human communication and computer commands. However, significant challenges remain in making such systems both efficient and practical for real-world usage.

The terminal command line interface (CLI) is a powerful tool widely used by developers and system administrators due to its flexibility and efficiency in managing resources. Nonetheless, the CLI presents a steep learning curve for non-technical users, requiring memorization of complex commands and syntax. This high threshold excludes many users from accessing the potential of terminal commands, creating a barrier for novices and non-technical individuals alike. Addressing this gap, we propose **XiaoFish**, a novel system that allows users to execute terminal commands through natural language. By enabling intuitive communication with the terminal via everyday language, XiaoFish aims to lower the entry barrier for terminal usage and enhance the overall user experience.

This work builds upon the concept of natural language interfaces (Lauriola and Moschitti, 2022) and leverages the Transformer architecture (Vaswani et al., 2023), a neural network that excels at capturing long-distance dependencies in sequences. By applying these technologies, XiaoFish is designed to convert natural language inputs into executable Bash commands, enabling users to perform complex terminal tasks without prior knowledge of command-line syntax. This project represents a significant step toward democratizing access to CLI tools, making them accessible and intuitive for all users.
The contributions of this paper are following:

1. We present a comprehensive system for natural language-to-command translation, utilizing an extended version of the NL2Bash dataset and a custom Transformer-based model.

2. We introduce novel features to test performance of model.

3. We provide a robust evaluation of our approach using nl2bash dataset, highlighting the challenges and potential of this emerging field.

The implementation and datasets used in this work are publicly available at `https://github.com/jchen3031/XiaoFish`. By integrating advanced NLP technologies with a user-centric design, XiaoFish aims to transform how users

interact with terminal commands, paving the way for broader adoption of command-line tools across diverse user groups.

This paper is structured as follows. Section 2 reviews related work, including natural language interfaces and the Transformer architecture. Section 3 introduced the motivation of the paper, gives answer why it is necessary to have a XiaoFish system with some example cases. Section 4 introduced our method to preprocessing the data. Section 5 details the methodology, including model design, and implementation strategies, detailed explanation of the components used in the model and their purpose. Section 6 introduced our experiment setting, with our novel test method and BLEU. Section 7 introduces our future plan for this project.

2 Related work

In this section, we will introduce our related work

2.1 NL2Bash

Lin et al. (2018) proposed nl2bash dataset, is a corpus designed for the task of translating natural language (NL) commands into corresponding Bash shell commands. This dataset is a valuable resource for natural language processing (NLP) and semantic parsing, particularly in domains requiring command-line automation. Below are the key characteristics of the dataset:

- **Diverse Set of Bash Utilities and Flags:**
  - Contains 102 unique Bash utilities (e.g., `ls`, `grep`, `cat`), covering a wide range of command-line operations.
  - Includes 206 unique flags (e.g., `-l`, `--help`, `-v`), providing flexibility and functionality to commands.
  - 15 reserved tokens are used as placeholders for semantic arguments, enabling abstraction and generalization.
- **Dataset Size:** Approximately 10,000 pairs of natural language descriptions and corresponding Bash commands are included, facilitating sequence-to-sequence learning and semantic parsing research.
- **Data Splitting:** The dataset is split into *train*, *dev*, and *test* sets. The splits are structured such that neither a natural language description nor a Bash command appears in more than one split, ensuring robust evaluation and preventing data leakage.
- **Command Templates:** A *command template* is defined as a Bash command with all its arguments replaced by their *semantic types* (e.g.,

filenames, directory paths, or numbers). This abstraction emphasizes reusable patterns and simplifies analysis.

2.2 NLP interface

Natural language processing (NLP) interface refers to the interface that uses natural language as input or output in human-computer interaction. It enables users to communicate with computer systems through everyday language without having to master and remember any programming languages and commands. The core of the NLP interface is to enable the model to understand natural language, so that users can complete the interaction more conveniently and efficiently. (Lauriola et al., 2021) The primary objective of NLP interfaces is to bridge the gap between human communication and computer systems, making interactions more intuitive and user-friendly. The core advantages of NLP interfaces include:

- **Ease of Use:** Users can express their needs in natural language, reducing the cognitive load of learning and recalling formal commands or programming languages.
- **Increased Accessibility:** NLP interfaces make technology accessible to non-technical users, enabling them to interact with complex systems effortlessly.
- **Efficiency and Flexibility:** Users can accomplish tasks through conversational interactions, often in fewer steps compared to traditional interfaces.
- **Enhanced User Experience:** By understanding and responding in human-like ways, NLP interfaces provide a more engaging and satisfying user experience.

Challenges in NLP Interfaces

Despite their numerous advantages, NLP interfaces face significant challenges:

- **Ambiguity:** Natural language is inherently ambiguous, with words and phrases often having multiple meanings depending on context.
- **Error Handling:** Errors in speech recognition, language understanding, or intent classification can lead to incorrect responses or task failures.
- **Domain Adaptation:** NLP models trained on general datasets may struggle to perform well in domain-specific scenarios.
- **Privacy and Security:** NLP interfaces often process sensitive user data, raising concerns about privacy, data security, and ethical use.

- **Multimodal Interaction:** Integrating natural language with other input/output modalities (e.g., images, gestures) adds complexity to the system design.

2.3 Transformer

Vaswani et al. (2023) proposed a new neural network structure called Transformer. The design of Transformer abandons the calculation of sequence order in traditional recurrent neural networks (such as RNN and LSTM), and is completely based on the self-attention mechanism, which enhances the model's ability to model long-distance sequence dependencies. In addition, based on the self-attention mechanism, Transformer can flexibly handle sequences of different lengths, and Transformer can better reason based on contextual content. The Transformer architecture is primarily composed of an encoder-decoder structure, where each layer of the encoder and decoder is built using multi-head self-attention and position-wise feed-forward networks.

Advantages of Transformer

The Transformer architecture brings several advantages compared to traditional sequence models like RNNs and LSTMs:

- **Parallelism:** Unlike RNNs, which process sequences sequentially, Transformer can process all tokens in a sequence simultaneously, significantly improving computational efficiency.
- **Modeling Long-Range Dependencies:** The self-attention mechanism allows the model to consider relationships between any two tokens in the sequence, making it particularly effective for long sequences.
- **Flexibility with Sequence Lengths:** The positional encoding enables the Transformer to handle sequences of varying lengths without additional modifications.
- **Scalability:** Transformer models can scale to billions of parameters (e.g., BERT, GPT) and perform well on a wide range of NLP tasks.

Impact and Extensions

Since its introduction, the Transformer has become the backbone of most state-of-the-art NLP models. Notable extensions and advancements based on the Transformer include:

- **BERT (Bidirectional Encoder Representations from Transformers):** A pre-trained Transformer-based model that excels in understanding bidirectional context for tasks such as question answering and text classification.

- **GPT (Generative Pre-trained Transformer):** A decoder-only Transformer designed for text generation, powering applications like ChatGPT.
- **T5 (Text-to-Text Transfer Transformer):** A model that frames all NLP tasks as text-to-text problems, enabling unified solutions for tasks like translation, summarization, and classification.
- **Vision Transformer (ViT):** Extends the Transformer architecture to computer vision tasks, where image patches are treated as tokens.
- **Reformer and Longformer:** Variants designed to handle very long sequences more efficiently by reducing the computational cost of self-attention.

The Transformer has revolutionized not only NLP but also other domains like computer vision and bioinformatics, showcasing its versatility and scalability. As research progresses, more efficient and specialized variants of the Transformer continue to emerge, further expanding its impact across diverse applications.

3 Motivation

Our motivation stems from the desire to remove these barriers and expand the availability of CLI environments to a more diverse audience. Unlike graphical interfaces, which often trade flexibility for simplicity, CLIs are still highly efficient but not intuitive enough for the average user. This work envisions a system where natural language serves as a middleman, allowing users to express their intent without having to master formal command-line syntax.

Furthermore, as computing systems become increasingly complex, the need for accessible and intuitive interfaces grows. Natural language understanding (NLU) has matured to the point where it can be applied to this problem space, providing an opportunity to rethink how users interact with terminal commands. However, this shift requires addressing challenges such as ambiguity in natural language, maintaining system efficiency, and ensuring reliable command translation.

By introducing XiaoFish, this work aims to empower users by making terminal commands accessible through simple, everyday language. Our system not only lowers the barrier to entry for using terminal environments, but also redefines CLI as a tool for everyone, regardless of their technical background. This vision aligns with the broader goal of democratizing technology and ensuring its benefits reach a wider audience. We will present several use cases where the XiaoFish system is needed.

Use Case: System Process Management for Administrators

System administrators often need to manage processes on servers to ensure smooth and uninterrupted operation. Consider the following scenario: a system administrator encounters zombie processes that consume system resources unnecessarily. Previously, the administrator needed to search for instructions on how to clean up zombie processes or memorize the corresponding complex command. With the **XiaoFish** system, this task becomes significantly simpler. The administrator can now simply enter the natural language query:

> *Clean up all zombie processes by instantly killing their parent process with the SIGKILL signal.*

The **XiaoFish** system interprets this query and generates the corresponding command:

```
kill -9 $(ps -A -ostat,ppid | grep -e '[zZ]' | awk '{ print $2 }')
```

This command identifies all zombie processes in the system, extracts their parent process IDs, and sends the `SIGKILL` signal to terminate them. By automating this process, **XiaoFish** significantly reduces the effort required for process management, making it easier and more efficient for system administrators to maintain server health.

Use Case: File Integrity Verification for Developers

Developers often need to ensure the integrity of their code files, especially when working on collaborative projects or managing large codebases. Consider the following scenario: a developer wants to verify the combined integrity of all Python files in a directory. Previously, the developer needed to search online for instructions on how to compute a hash for multiple files or memorize the corresponding command. With the **XiaoFish** system, this task becomes much more straightforward. The developer can simply enter the natural language query:

> *Calculate the md5 sum of all *.py files in the current directory*

The **XiaoFish** system interprets this query and generates the corresponding command:

```
cat *.py | md5sum
```

4 Data Acquisition and Preprocessing

4.1 Introduction

To achieve natural language control of the operating system, our goal is to allow any user to perform tasks on their computer for a long period and realize the manipulation of the corresponding commands by simply stating their purpose in natural language. Due to the uniqueness of the command data, there are many challenges in the construction of the dataset, such as the irregularity of the command syntax, the wide coverage, and the large number of invisible words. We use the NL2Bash corpus, which is a collection of commonly used Bash commands from websites such as $Q\&A$ forums, technical blogs, and course materials, and careful quality control yielded more than 9,000 English command pairs covering more than 100 unique Bash utilities. Table 1 shows a few examples of the high-quality corpus (Lin et al., 2018)

Table 1: Example natural language descriptions and the corresponding shell commands from NL2Bash

Natural Language	**Bash Commands**

Find java files in the current directory tree that contain the pattern 'TODO' and print their names	`grep -l "TODO" *.java`
	`find . -name "*.java" -exec grep -il "TODO" {}`
	`find . -name "*.java" | xargs -I {} grep -l "TODO" {}`

Display the 5 largest files in the current directory and its sub-directories	`find . -type f | sort -nk 5,5 | tail -5`
	`du -a . | sort -rh | head -n5`
	`find . -type f -printf '%s %p\n' | sort -rn | head -n5`

Search for all jpg images on the system and archive them to tar ball "images.tar"	`tar -cvf images.tar $(find / -type f -name *.jpg)`
	`tar -rvf images.tar $(find / -type f -name *.jpg)`
	`find / -type f -name "*.jpg" -exec tar -cvf images.tar {}`

The corpus is composed of textual command pairs, each pair consisting of Bash commands crawled on the Web and expert-generated natural language descriptions (`https://github.com/TellinaTool/nl2bash/tree/master/data`.).

4.2 Corpus Statistics

The NL2Bash dataset is a dataset of 9305 natural language descriptions with Bash command pairs, which covers 102 unique Bash tools and 206 flags.The average length of Bash commands is 7.7 tokens, which is relatively short for natural language. The median frequency of both word frequency and command tokens is 1, which is due to the fact that a large number of open vocabulary words (dates, filenames, etc.) appear only 1 time in the corpus. Figure 1 shows the 50 most common Bash tools for the dataset species and their frequency distribution, which shows that the data has a long-tailed character, with the most frequently used find tool appearing 6,268 times, followed by xargs with 1,047 occurrences. The least frequent Bash tool appeared only 984 times, reflecting the uneven distribution of tool usage in the dataset (Zettlemoyer and Collins, 2012).

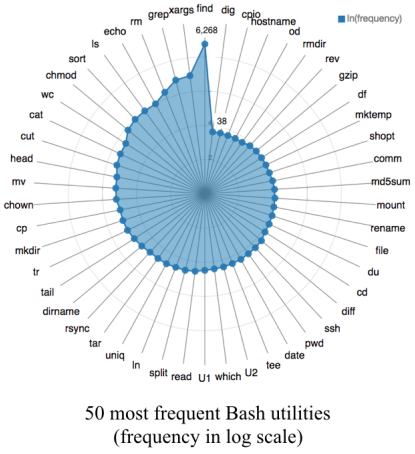

Figure 1: Top 50 most frequent bash utilities in the dataset with their frequencies in log scale

4.3 Preprocessing Techniques

The processing of the NL2Bash dataset focuses on the steps of cleaning, de-duplication, noise reduction, merging, and textual feature extraction, providing data consistency and usability. Through reasonable preprocessing steps, we achieved standardization and structuring of the data, laying the foundation for the subsequent model training task. The NL2Bash dataset consists of command and natural language pairs, and the two parts of each data pair are associated through ID fields. We processed the dataset through ID fields to further improve its quality and extract key features for subsequent model training.

4.4 Data Preprocessing Steps

1. **ID Field Cleaning** To ensure the matching of ID fields in the Commands and Descriptions subsets:
    - Convert all IDs to lowercase to eliminate case differences.
    - Remove spaces in ID fields to avoid matching errors caused by blank characters.
2. **Data De-Redundancy** To reduce data redundancy and improve consistency:
    - De-duplicate the ID fields in the dataset, keeping only the first record corresponding to each ID.
    - This reduces interference from duplicate records during model training and analysis.
3. **Text Noise Reduction** For the text data:
    - **Commands**: Use regular expressions to normalize and replace sensitive information:
        - Replace IP addresses with *<IP_ADDRESS>*.
        - Replace file paths with *<PATH>*.
        - Replace numeric values with *<NUM>*.
    - **Descriptions**: Remove brackets and bracketed content to reduce unnecessary noise.
4. **Data Collection and Tagging with Missing Values** Merge the Commands and Descriptions subsets through an outer join to preserve all commands and their descriptions. Fill missing data with *<MISSING>* to ensure data completeness and labeling clarity.
5. **Feature Extraction** Extract key features from the description text using the TF-IDF (Term Frequency-Inverse Document Frequency) method. This effectively highlights the core content of the description text and filters out noise words.

4.5 Examples of preprocessing data

The final format of the data as a result of the pre-processing process described above is as follows 2:

Table 2: Example of a pre-processed data pair

ID	Cleaned Command	Cleaned Description	Description Keywords
example_123	`ping <IP_ADDRESS> <FLAG> <NUM>`	Send 4 packets to a specific IP address	send packets specific address
example_456	`grep <FLAG> [pattern] <FILE>`	Search for case-insensitive errors in the system log file	search case-insensitive errors log
example_789	`find <PATH> <FLAG> [pattern]`	Find all text files in the current directory and subdirectories	find text files directory subdirectories
example_101	`tar <FLAG> <OUTPUT> [pattern]`	Create a tar archive named archive.tar containing all PNG files	create tar archive containing png

5 Methodology

In this section, we will introduce our core Transformer-based deep learning model **XiaoFish** This model is designed to process sequential data efficiently and is tailored for tasks that require attention mechanisms, such as translation, sequence prediction, or classification. Below is a comprehensive explanation of each component in the model, detailing its purpose, implementation, and functionality.

5.1 Positional Encoding

Positional encoding is an essential component of the transformer model, designed to provide the model with information about the order of tokens in the input sequence. Unlike recurrent neural networks (RNNs), which inherently process data sequentially, transformer architectures process all tokens simultaneously, making it necessary to explicitly encode positional information. Positional encoding achieves this by introducing unique representations for each position in the sequence, allowing the model to distinguish between tokens at different positions and capture the sequential nature of the data.

A well-designed positional encoding scheme ensures that the model can effectively leverage both positional and content-based information during training. The encoding uses a combination of sine and cosine functions of varying frequencies to generate a fixed set of positional embeddings. These

embeddings are added to the input token embeddings, ensuring that the positional information is seamlessly integrated into the model's attention mechanism. This approach not only preserves the order of the sequence but also allows the model to generalize across sequences of varying lengths.

The flexibility and mathematical elegance of positional encoding make it particularly suited for transformer-based architectures, enabling the model to capture long-range dependencies and process ordered data effectively. The following **Algorithm 1** provides the pseudocode for our positional encoding. The Key Details of this algorithm is following:

Algorithm 1 Positional Encoding

Require: *position* (sequence length), *d_model* (model dimensionality)
Ensure: *pos_encoding* (positional encoding matrix)

Step 1: Compute angle rates:
For each position *pos* and dimension i:
    Compute the angle rate as:

$$\text{angle\_rate} = \frac{1}{10000^{\frac{2\cdot(i//2)}{d\_model}}}$$

Multiply each position *pos* by the angle rate to generate an angle matrix.
Step 2: Apply sine and cosine functions:
For even indices i in the dimension (e.g., $i = 0, 2, 4, \dots$):
    Apply the sine function:

$$\text{angle\_rads}[i] = \sin(\text{angle\_rads}[i])$$

For odd indices i in the dimension (e.g., $i = 1, 3, 5, \dots$):
    Apply the cosine function:

$$\text{angle\_rads}[i] = \cos(\text{angle\_rads}[i])$$

Step 3: Expand dimensions:
Add an extra dimension to the computed matrix to prepare it for batch processing.
Step 4: Convert data type:
Cast the positional encoding matrix to float32 for compatibility with TensorFlow operations.
Return: The final positional encoding matrix *pos_encoding*.

- The encoding alternates between sine and cosine functions for even and odd indices, ensuring that each dimension of the encoding has a unique

frequency.

- The generated matrix has a shape of (1, position, d_model) and is added to the input embeddings.

5.2 Multi-Head Attention (MHA)

The multi-head attention mechanism is a core component of the Transformer model, which can efficiently process sequence data and capture the associations between different parts of the sequence. By using multiple attention heads, the model can focus on the input data in different subspaces, thereby capturing local and global dependencies simultaneously. This design allows the model to consider interactions between different positions, which is particularly important for tasks that require context-aware predictions.

A well-designed multi-head attention mechanism can help the model achieve a good balance between extracting local detail features and understanding the overall context. Each attention head independently calculates the attention score of the input sequence and combines the results to provide a more comprehensive input representation. This approach allows the model to process data from various perspectives while ensuring computational efficiency. The final output is obtained by concatenating the results of all attention heads and mapping them back to the target dimension.

The core advantage of the multi-head attention mechanism is its flexibility. It can focus on different parts of the sequence simultaneously, thereby improving the model's ability to understand long-range dependencies and contextual relationships. This feature makes it perform well in many tasks that require complex context modeling. The following **Algorithm 2** provides the pseudocode for our MHA

Through this algorithm we enable the model to focus on different parts of the input sequence simultaneously and capture both short-term and long-term dependencies.

5.3 Feed-Forward Network (FFN)

The feed-forward network (FFN) is a crucial component of the transformer architecture, responsible for introducing non-linear transformations to the data after the attention mechanism. Unlike the attention layers, which focus on capturing dependencies between tokens, the FFN operates independently on each token, applying the same set of transformations regardless of their positions in the sequence. This design ensures that the model can enhance its representational capacity while maintaining computational efficiency.

A well-structured FFN consists of two fully connected layers separated by a non-linear activation function, such as ReLU. The first layer expands the dimensionality of the input, allowing the network to learn richer intermediate representations, while the second layer projects the data back to the original

Algorithm 2 Multi-Head Attention Mechanism

Require: *d_model* (embedding dimension), *num_heads* (number of attention heads), v, k, q (value, key, query matrices), *mask* (optional masking matrix)
Ensure: Multi-head attention output
Initialize:
Compute depth per head: depth = d_model/num_heads
Define learnable weight matrices for queries (W_q), keys (W_k), values (W_v), and output (W_o), each of size $d\_model \times d\_model$
Step 1: Linear projections
Compute projected queries: $Q = q \cdot W_q$
Compute projected keys: $K = k \cdot W_k$
Compute projected values: $V = v \cdot W_v$
Step 2: Reshape for multi-heads
Reshape Q, K, and V to dimensions (batch_size, num_heads, sequence_length, depth)
Step 3: Scaled Dot-Product Attention
Compute attention scores: scores = $Q \cdot K^T / \sqrt{\text{depth}}$
Apply masking (if provided): scores+ = (mask $\times -10^9$)
Normalize scores using softmax: attention_weights = softmax(scores, axis $= -1$)
Compute attention output: output = attention_weights $\cdot V$
Step 4: Concatenate and Project
Transpose and reshape output to (batch_size, sequence_length, d_model)
Pass through the final linear layer: final_output = output $\cdot W_o$
Return: final_output

dimensionality. This combination of expansion and projection enables the FFN to model complex interactions within the data, contributing to the overall flexibility and power of the transformer model.

By applying the FFN independently to each token, the model gains the ability to learn diverse features without being constrained by token-to-token dependencies. This independence complements the global contextual understanding provided by the attention mechanism, resulting in a balanced and effective architecture. The following **Algorithm 3** provides the pseudocode for our feed-forward network. We use two dense layers, and the first layer uses

Algorithm 3 Point-Wise Feed-Forward Network

Require: d_model (embedding dimension), dff (dimensionality of the feed-forward layer)
Ensure: Feed-forward network function
Step 1: Define a two-layer feed-forward network.
First layer:
    Fully connected layer with dff units and ReLU activation.
Second layer:
    Fully connected layer with d_model units (no activation).
Step 2: Use a sequential model to combine the layers.
The network processes inputs sequentially through the two layers.
Return: The feed-forward network function.

ReLU activation to achieve wireless. In addition, in general, dff is usually larger than d_model, which helps to increase the network capacity.

5.4 Learning Rate Scheduler

In the transformer model, learning rate scheduler is defined to help the model converge effectively during training. A good learning scheduler is very important for the successful training of the model. If the learning rate used is too small, this will cause the model to update very small steps each time, resulting in a large number of training epochs, which means more training time. However, if the learning rate used is too large, this will cause the model to update too large each time, leading to overfitting problems or inability to continue learning. Therefore, a good learning rate scheduler is designed to enable the model to use different learning rates to learn in different epochs to avoid inappropriate learning rates affecting model training. Generally speaking, the learning rate should be relatively large at the beginning of training to allow the model to quickly approach the appropriate weights, and then the learning rate should gradually decrease, in order to allow the model to learn slowly and avoid over-optimization all at once. The following **Algorithm 4** is the pseudo code of our learning rate scheduler

Algorithm 4 Custom Learning Rate Schedule

Require: *d_model* (embedding dimension), *warmup_steps* (default: 4000), *step* (current training step)
Ensure: Learning rate value for the current step
1: **Step 1: Initialize parameters:**
2: Cast *d_model* to float32 for precision.
3: Set the number of warmup steps to *warmup_steps*.
4: **Step 2: Compute learning rate for the current step:**
5: Compute the inverse square root of the current step: $\text{arg1} = \frac{1}{\sqrt{\text{step}}}$
6: Compute the scaled warmup factor: $\text{arg2} = \text{step} \cdot (\text{warmup\_steps}^{-1.5})$
7: Compute the learning rate as:

$$\text{learning\_rate} = \frac{1}{\sqrt{d\_model}} \cdot \min(\text{arg1}, \text{arg2})$$

8: **Step 3: Return the computed learning rate.**

Explanation of Learning Rate Updates

The learning rate schedule defined by the `CustomSchedule` class follows a dynamic update strategy:

- Initially, during the warm-up phase, the learning rate increases linearly with the training step. This helps the model to stabilize its learning during the early stages.
- After the warm-up phase, the learning rate decreases proportionally to the inverse square root of the training step. This allows the optimizer to take smaller steps as training progresses, avoiding overshooting the optimal solution.
- The overall formula is given by:

$$\text{Learning Rate} = \frac{1}{\sqrt{d\_model}} \cdot \min\left(\frac{1}{\sqrt{\text{step}}}, \frac{\text{step}}{\text{warmup\_steps}^{1.5}}\right)$$

- The parameter *d_model* scales the learning rate inversely with the embedding size, ensuring compatibility with larger models.

This schedule is widely used in transformer-based architectures, such as the original Transformer model, to balance exploration (large steps early) and exploitation (small steps later).

5.5 Overall Transformer Architecture

The XiaoFishBot Transformer architecture includes:

- Encoder: Processes input sequences with self-attention and feed-forward networks.
- Decoder: Generates output sequences, attending to both the input sequence and previous outputs.

Each layer includes:

1. Multi-head attention
2. Add and Norm (residual connections and layer normalization)
3. Feed-forward network

Our XiaoFish model has a total of **5.7M** parameters. It uses 4 number of layers, 8 number of heads, 128 dimension of model and 512 dimension of feed forward network. The graph of model is the Figure 2

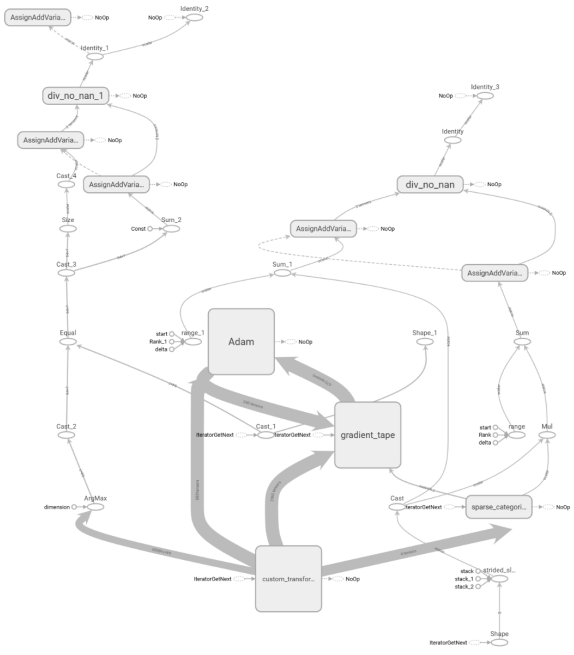

Figure 2: XiaoFish model graph

6 Experiment Design

6.1 Evaluation Methodology

To comprehensively evaluate the performance of our Transformer model, we adopted two distinct approaches: BLEU scoring and functionality testing. These methods were chosen to complement each other, addressing the limitations inherent in each individual approach while providing a holistic view of the model's performance.

6.2 BLEU Score Evaluation

The Bilingual Evaluation Understudy (BLEU) score, introduced by Papineni et al. (2002), is a widely used metric for evaluating the quality of text generated by models in natural language processing tasks, such as machine translation, text summarization, and caption generation. BLEU measures the lexical similarity between a model's generated output and one or more reference outputs by analyzing n-grams, which are contiguous sequences of words. In this approach, BLEU was employed to assess how closely the model's predictions align with the reference sequences, providing a standardized quantitative measure of performance. The BLEU evaluation process begins by tokenizing both the predicted and reference sequences. For each n-gram size n, precision p_n is calculated as the ratio of n-grams in the predicted output that match those in the reference. The overall BLEU score is then computed as the geometric mean of these n-gram precisions, weighted equally or according to pre-defined weights W_n . The formula for BLEU is:

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

Where:

- p_n is the precision for n-grams of size n,
- W_n represents the weight assigned to each n-gram precision,
- BP is the brevity penalty to penalize shorter outputs.

The brevity penalty BP is calculated as follows:

$$\textit{Brevity Penalty} = \begin{cases} 1, & \text{if } c > r \\ e^{\left(1-\frac{r}{c}\right)}, & \text{if } c \leq r \end{cases}$$

Where:

- c is the length of the candidate translation,
- r is the effective reference length.

The brevity penalty discourages models from generating overly short predictions. It ensures that the predicted output length c is proportional to the reference length r. For our model evaluation, BLEU was calculated up to 4-grams (N = 4) with uniform weights $W_n = \frac{1}{N}$. The n-gram precisions are computed as:

$$p_n = \frac{\textit{Number of matching } n_{grams}}{\textit{Total number of } n_{grams} \textit{ in the candidate}}$$

To address potential zero values in higher-order n-gram precision, especially for shorter sequences, smoothing techniques were applied using the NLTK library. Smoothing adjusts the precision values to avoid extreme penalties for sequences that lack matches in higher-order n-grams, resulting in more robust BLEU calculations. This is particularly important for tasks where the output sequence length or diversity can vary significantly.

While BLEU provides a robust and standardized evaluation metric, it has inherent limitations. The metric measures only surface-level token overlap and does not account for semantic equivalence. For instance, BLEU may penalize correct outputs that use synonyms or paraphrased structures. Despite this, it remains a cornerstone for syntactic evaluation in tasks emphasizing lexical fidelity, such as machine translation Papineni et al. (2002); Banerjee and Lavie (2005).

By integrating the BLEU score into the evaluation framework, we make our evaluation a reproducible and interpretable metric while mitigating its limitations through complementary evaluation approaches.

6.3 Functionality Testing

Functionality testing is a task-specific evaluation approach designed to measure the practical performance of the model in real-world scenarios. Unlike surface-level metrics such as BLEU, functionality testing focuses on whether the model's output meets the functional requirements of the task. This approach employed functionality testing to complement BLEU scoring, enabling a comprehensive evaluation of the model's performance.

The evaluation process involves comparing the model's predictions against ground truth outputs to determine their correctness. The primary metric used in functionality testing was accuracy, defined as the proportion of correctly predicted outputs over the total number of test cases. The formula for calculating accuracy for a single test case is:

$$\textit{Accuracy} = \frac{\textit{Number of Correct Predictions}}{\textit{Total number of Predictions}}$$

For a dataset containing M test cases, the average accuracy is computed as:

$$Average\ Accuracy = \frac{1}{M} \sum_{i=1}^{M} Accuracy_i$$

where $Accuracy_i$ is the accuracy for the i-th test case.

To ensure a rigorous comparison of predicted outputs against ground truth, functionality testing was conducted under controlled conditions. Specifically, for each command predicted by the model, a subprocess was forked in Ubuntu to execute the command and obtain the corresponding output. This output was then directly compared to the expected target result. By evaluating the execution results under the same environment and conditions, this methodology eliminates variability introduced by external factors, ensuring that differences between predicted and target outputs are solely attributable to the model's performance. This subprocess-based execution also enables testing the correctness of the model in handling complex command structures or edge cases. Functionality testing also incorporates qualitative assessments to analyze the model's robustness. For instance, edge cases, such as ambiguous commands or noisy inputs, were included in the evaluation pipeline. These tests provided insights into the model's ability to generalize and its behavior under less common but critical scenarios. Observations from these tests highlighted areas where the model excels, such as handling well-structured inputs, and areas that require further optimization, such as managing highly variable or corrupted inputs. This layer of qualitative testing ensures that the evaluation captures not only accuracy but also the robustness of the system.

The flexibility of functionality testing allows it to be tailored to the specific requirements of the task. For example, in our evaluation, accuracy was supplemented with qualitative observations to identify patterns and limitations in the model's predictions. These observations provided insights into areas where the model excels, such as handling well-structured inputs, as well as areas where improvements may be needed, such as dealing with ambiguous or noisy inputs. Additionally, functionality testing ensures that models deliver outputs that are meaningful in practical applications, even when they deviate from syntactic fidelity.

While functionality testing offers significant advantages in evaluating practical performance, it is inherently task-specific and requires careful design to ensure the test cases are representative of the target application. Furthermore, the lack of standardization in functionality testing can limit its comparability across different studies. Nevertheless, when combined with standardized metrics like BLEU, functionality testing provides a more holistic and nuanced evaluation framework. By integrating functionality testing into the evaluation pipeline, this evaluation ensures that the model's outputs are not only syntactically accurate but also meaningful and contextually appropriate. By integrating functionality testing into the evaluation pipeline,

this evaluation ensures that the model's outputs are not only syntactically accurate but also meaningful and contextually appropriate. Together with BLEU scoring, functionality testing enables a comprehensive assessment of the model's performance, addressing both its strengths and limitations.

6.4 Results and Analysis

The evaluation of the model was conducted using two complementary metrics: BLEU score and accuracy, to provide a comprehensive assessment of its performance. The results were obtained on test sets of varying sizes, specifically 100, 200, 500, and 1000 samples, and are presented in Table 3. These two evaluation methods ensure that the model's outputs are measured both syntactically and functionally.

The BLEU score, which quantifies the lexical overlap between the model's predictions and reference sequences, remained constant at **81.70%** across all test set sizes. This consistency is primarily due to the fact that the same dataset was used for both training and evaluation. As a result, the model demonstrates a high degree of syntactic fidelity when generating outputs for sequences it has already seen, leading to stable BLEU scores across different evaluation scales.

In terms of accuracy, the model demonstrated strong functional performance across all dataset sizes, with minimal variations. For smaller test sets, such as 100 and 200 samples, the model achieved an accuracy of **91.00%**, reflecting its ability to deliver highly reliable predictions. As the test size increased to 500 and 1000 samples, the accuracy exhibited a slight decrease to **88.40%** and **89.30%**, respectively. This marginal decline suggests that larger datasets may introduce more challenging or diverse input scenarios. However, the accuracy values remain high, indicating that the model continues to perform robustly and effectively even as the test scale increases.

Table 3: Evaluation Results

Size	Pass	Accuracy	BLEU
100	91	91.00%	81.70%
200	**182**	**91.00%**	**81.70%**
500	442	88.40%	81.70%
1000	**899**	**89.90%**	**81.70%**

The alignment between accuracy values and the number of passed predictions further reinforces the consistency of the evaluation. For instance, in the test set of 1000 samples, 893 predictions were correct, resulting in an accuracy of **89.30%**. Similarly, for a test set size of 100, 91 correct predictions yielded an accuracy of **91.00%**. This consistency between the absolute pass

counts and the calculated accuracy highlights the reliability of the evaluation methodology. The result shows in Figure 3.

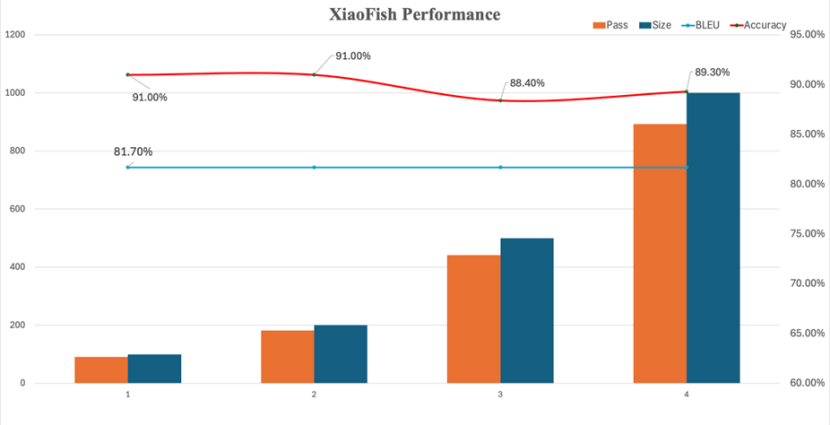

Figure 3: Graph of the Results

Overall, the results demonstrate that the model achieves stable performance across varying dataset sizes. The consistency of the BLEU score underscores the model's ability to produce syntactically faithful outputs, while the high accuracy values confirm its functional correctness. Together, these results validate the model's robustness and suitability for real-world applications, as it maintains reliable performance across both syntactic and functional evaluation criteria.

7 Future Work

The current challenge of our model is that once the input natural language contains words that have not appeared in the dataset, the performance of the model will be significantly reduced. To solve this problem, we can pre-train the model on a larger corpus in the future, and then fine-tune it on the nl2bash dataset. In addition, exploring different optimizers or more complex attention mechanisms may further improve performance. We will use different models and different datasets to compare with our method. In addition, due to the way the loss is calculated and the poor performance of traditional accuracy evaluation on NLP problems, we also plan to customize the loss function. At present, a possible effective idea is to use our method to evaluate the accuracy during training, and then multiply it by a specific weight and add it to the original loss function.

8 Conclusion

In summary, our developed XiaoFish system, and it shows very good performance on the nl2bash dataset. Although there are still many challenges in real world example, we believe this is a good start to show the potential of natural language processing in the field of command interaction.

References

Banerjee, S. and Lavie, A. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Goldstein, J., Lavie, A., Lin, C.-Y., and Voss, C., editors, *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Dahl, D. A., Bates, M., Brown, M. K., Fisher, W. M., Hunicke-Smith, K., Pallett, D. S., Pao, C., Rudnicky, A. I., and Shriberg, E. E. (1994). Expanding the scope of the atis task: The atis-3 corpus. *Proceedings of the workshop on Human Language Technology*, pages 43–48.

Desai, S., Huang, J., Quirk, C., and Galley, M. (2016). Program synthesis using natural language. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 33–41.

Dijkstra, E. W. (1978). On the foolishness of 'natural language programming'. In *International Federation for Information Processing*, pages 1–7.

Lauriola, I., Lavelli, A., and Aiolli, F. (2021). An introduction to deep learning in natural language processing: Models, techniques, and tools. *Neurocomputing*, 470:443–456.

Lauriola, I. and Moschitti, A. (2022). An introduction to natural language interfaces. *ACM Computing Surveys*, 55(4):1–35.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. (2018). Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Quirk, C., Mooney, R. J., and Galley, M. (2015). Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 878–888.

Sammet, J. E. (1966). *Programming languages: History and fundamentals*. Prentice-Hall.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

Wilensky, R., Chin, D., Luria, M., Martin, J., Mayfield, J., and Wu, D. (1984). The unix consultant project. *Computational Linguistics*, 14(4):35–84.

Wilensky, R., Chin, D., Luria, M., Martin, J., Mayfield, J., and Wu, D. (1988). Relevance and multiple theories of discourse. *Computational Linguistics*, 14(2):17–29.

Zettlemoyer, L. S. and Collins, M. (2012). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars.