# Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models

**Yuqi Zhu[1], Jia Li[2], Ge Li[*2], YunFei Zhao[2], Jia Li[2], Zhi Jin[2], Hong Mei[2,3]**

[1]Academy of Military Sciences, China

[2]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China

[3]Advanced Institute of Big Data, Beijing, China

zhuyuqi1997@126.com, lijia@stu.pku.edu.cn, {lige, zhaoyunfei, lijiaa, zhijin, meih}@pku.edu.cn

## Abstract

Recently, Large Language Models (LLMs) have shown impressive abilities in code generation. However, existing LLMs' decoding strategies are designed for Natural Language (NL) generation, overlooking the differences between NL and programming languages (PL). Due to this oversight, a better decoding strategy for code generation remains an open question. In this paper, we conduct the first systematic study to explore a decoding strategy specialized in code generation. With an analysis of loss distributions of code tokens, we find that code tokens can be divided into two categories: challenging tokens that are difficult to predict and confident tokens that can be easily inferred. Among them, the challenging tokens mainly appear at the beginning of a code block. Inspired by the above findings, we propose a simple yet effective method: Adaptive Temperature (AdapT) sampling, which dynamically adjusts the temperature coefficient when decoding different tokens. We apply a larger temperature when sampling for challenging tokens, allowing LLMs to explore diverse choices. We employ a smaller temperature for confident tokens avoiding the influence of tail randomness noises. We apply AdapT sampling to LLMs with different sizes and conduct evaluations on two popular datasets. Results show that AdapT sampling significantly outperforms state-of-the-art decoding strategy.

## Introduction

Code generation aims to automatically generate a program that satisfies a natural language requirement (Li et al. 2023d,a,b). In recent years, Large Language Models (LLMs) have attracted great attention for their potential for automating coding (Li et al. 2023c,e). Noteworthy models like AlphaCode (Li et al. 2022) and Codex (Chen et al. 2021) have demonstrated their impressive ability to solve unforeseen programming challenges.

LLMs rely on a decoding strategy to generate code. Existing LLM's decoding strategies for code generation mainly fall into two categories. The first category is search-based methods, which aim to maximize the probability of the next generated token, including greedy search (Black and E 2012) and beam search (Freitag and Al-Onaizan 2017).

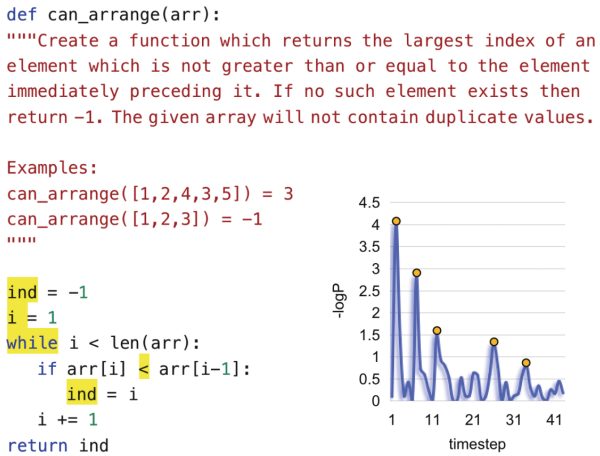


Figure 1: An illustration of a code snippet and its corresponding loss distribution. The challenging tokens are highlighted in yellow.

Nonetheless, the results generated by these methods lack diversity and are prone to generating empty or repetitive results (Zhang et al. 2021). The second category is sampling-based methods, which randomly sample the next token based on the probability distribution. The state-of-the-art (SOTA) approach (Chen et al. 2021) uses temperature sampling that reshapes the probability distribution by introducing a temperature coefficient to control the level of sampling randomness.

Despite the promising results, temperature sampling has limitations in code generation. Since it is initially used to generate Natural Language (NL) with a flexible syntax, its effectiveness decreases when transitioning to code generation, which is a high-accuracy demanding task. For instance, CodeGeeX (Zheng et al. 2023), when utilizing temperature sampling, attains 36.0% Pass@15 on the HumanEval dataset. Thus, it is necessary to explore more advanced decoding strategies to improve the accuracy of LLMs in code generation.

In this paper, we present the first systematic study to explore a decoding strategy for code generation with LLM. Our contributions can be summarized as follows:

**(1) By analyzing the loss distribution of code tokens,**

*Corresponding author

**we categorize code tokens into challenging tokens and confident tokens.** We design comparative experiments to investigate the differences in loss distributions between source code and NL text. Our analysis reveals that the source code suffers lesser variation in loss values during generation than NL text. Next, we visualize the loss distribution of source code. We find an apparent discrepancy in loss values among different code tokens. In this study, we refer to these tokens with high loss values as challenging tokens. With statistical analysis, we find that challenging tokens frequently appear at the beginning of a code block. An illustration of the challenging tokens is shown in Figure 1. The remaining tokens, characterized by low loss values for LLMs, are referred to as confident tokens.

**(2) In light of our findings, we propose Adaptive Temperature (AdapT) sampling, which dynamically adjusts the temperature coefficient.** Compared to standard temperature sampling, AdapT sampling dynamically adjusts the temperature coefficient $T$ according to the type of next-tokens. Our motivation is that LLMs require exploring diverse choices for challenging tokens. For confident tokens, we should select tokens with high probabilities. Specifically, for challenging tokens, AdapT sampling utilizes a high $T$ value to increase sample diversity. AdapT sampling uses a low $T$ value for confident tokens to minimize randomness noise.

**(3) Experimental results show that AdapT sampling can improve the pass@$k$ metric on HumanEval and MBPP datasets with different sizes of LLMs.** We apply the AdapT sampling to multiple LLMs with various sizes (from 2B to 13B) and conduct evaluations on two representative code generation datasets (HumanEval and MBPP). Experimental results show AdapT sampling outperforms the SOTA decoding strategy which uses a standard temperature sampling. For example, it surpasses the pass@15 metric over the SOTA method by up to 13.6% on HumanEval. We further investigate the robustness of AdapT sampling to different hyperparameter settings and the quality of the code generated by AdapT sampling. Our code is available at https://github.com/LJ2lijia/AdapT.

**(4) Future directions.** Based on our findings, we list the current challenges and propose future research directions on developing effective decoding strategies for code generation.

# Background

## Code Generation with LLMs

LLMs are transformer-based models that are trained using large corpora of NL text and source code. In recent years, LLMs have achieved impressive results in automatic code generation. Among LLMs, the GPT family of LLMs from OpenAI is popular and powerful, including GPT-3 (175B parameters) (Brown et al. 2020), Codex (175B parameters) (Chen et al. 2021), etc. Since OpenAI LLMs are closed-source, there have been many attempts to reproduce similar LLMs, such as CodeGen (Nijkamp et al. 2022a), CodeGeeX (Zheng et al. 2023), InCoder (Fried et al. 2022).

## Decoding Strategy

Given a requirement $x$, LLMs rely on a decoding strategy to generate the code auto-regressively. The decoding strategy determines how LLMs select the next token $y_t$ based on the context $y_{<t}, x$. $y_{<t}$ is the token sequence that has been generated. There has been a series of works exploring decoding strategies for NL generation, and these methods have been subsequently applied to code generation. They can be classified into two categories: search-based and sampling-based methods.

**Search-based Decoding Stratrgy** Greedy search (Mou et al. 2015) is one of the most commonly used decoding strategies. In greedy search, the model selects the next token which maximizes the probability: $y_t = \arg\max_y p(y_t|y_{<t}, x)$. Despite its simplicity, it may lead to overly conservative results and a lack of diversity (See et al. 2019). Beam search (Freitag and Al-Onaizan 2017) is an improved version of greedy search. This algorithm retains top $B$ (beam size) tokens with the highest probability. However, it has been found that beam search results in degenerations such as repetitions and empty (Holtzman et al. 2019).

**Sampling-based Decoding Strategy** The degenerations such as empty sequences and repetitions can be alleviated using sampling decoding, which randomly selects the next token based on predicted probability.

Temperature sampling (Ackley, Hinton, and Sejnowski 1985) has been applied widely, it uses a temperature coefficient $T$ (usually $\in [0, 1]$) to control the sampling randomness. Given the logits $u$ and temperature $T$, the softmax distribution $p'$ is re-estimated as:

$$p'(y_t|y_{<t}, x) = \frac{\exp(\frac{(u(y_t|y_{<t},x))}{T})}{\sum_{j=1}^{n} \exp(\frac{(u(y_j|y_{<t},x))}{T})} \quad (1)$$

In addition, researchers propose Top-$k$ (Fan, Lewis, and Dauphin 2018) sampling to further improve performance. At each step, Top-$k$ sampling filters the $k$ most probable next tokens and redistributes the probability among these $k$ tokens for sampling. However, the unreliable tail problem in the Top-$k$ sampling may affect the sampling quality. Top-$p$ sampling (Holtzman et al. 2019) eliminates the unreliable tail problem by sampling from the smallest token set whose cumulative probability reaches the threshold $p$.

LLMs usually use the method of combining temperature sampling and Top-$p$ sampling to achieve SOTA results (Chen et al. 2021; Nijkamp et al. 2022a; Fried et al. 2022). Specifically, the logits are first rescaled with temperature $T$. After this, Top-$p$ sampling is employed to derive the final results. Existing work (Chen et al. 2021) finds out that temperature coefficient $T$ has an obvious influence on the code generation results. Increasing the $T$ value can enhance the chance of exploring the correct answers, but this comes at the cost of introducing more errors in the generation results. Therefore there is a need to develop more effective sampling methods specifically for code generation.

statistical evaluation of all 40 samples

| metric | NL | code |
|---|---|---|
| Standard Deviation | 0.30 | 0.24 |
| Skewness Coefficient | 4.11 | 3.31 |
| Perplexity | 4.73 | 1.61 |

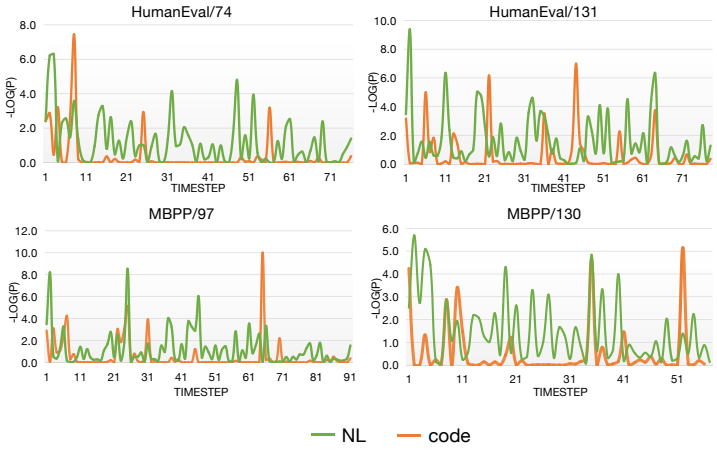


Figure 2: Comparison of loss distributions on NL text and source code. The left table shows the statistical results of different distributions on 40 samples. The right figures show several examples of loss distributions on NL text and source code.

## Analysis of the Code Generation Process

In this section, we first investigate the differences between NL generation with LLMs and code generation with LLMs. We compared NL text's loss distributions (i.e., cross-entropy loss (Brownlee 2019)) to ones of source code.

Furthermore, we analyze the fluctuations of loss values of code tokens within code snippets. We find that code tokens can be categorized into challenging tokens and confident tokens. Based on the analysis, we discuss the challenges encountered in code generation.

### Loss Distribution Comparison

In this section, we analyze the differences between the process of generating NL text and source code with LLMs. We choose loss distribution as the comparison metric.

We conduct experiments with a powerful LLM for source code - CodeGen. We select the CodeGen-mono with 2 billion parameters (CodeGen-2B) as the base model. This model is trained with a 635GB code corpus and 1159GB English text data. Therefore, CodeGen can generate NL text and code. The datasets used in this section are HumanEval (Chen et al. 2021) and MBPP (Austin et al. 2021), which are representative datasets in code generation.

First, we randomly select 20 code samples each from HumanEval and MBPP datasets. Then, we manually write NL descriptions for each code snippet, which describes the functionality of the code. We keep the alignment of the length of NL and code as much as possible while maintaining text fluency. As a result, we obtain 40 NL descriptions aligned with their corresponding code snippets. Next, we gather the loss values of the CodeGen model on these NL descriptions and code snippets, respectively.

We use various metrics (e.g. mean value (Runnenburg 1978), standard deviation, skewness, and perplexity) to compare the loss distributions of NL descriptions and source code. Standard deviation (Bland and Altman 1996) reflects the average amount of variability. Skewness (Brown 2011) is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. Perplexity (Brown et al. 1992) is a measurement of how confidently an LLM predicts a sample. As shown in the table in Figure 2, the average value of losses on NL descriptions is higher than the one on the source code. When compared with code, NL suffers greater variation in prediction loss during generation. The value of the skewness shows that there are more tokens with large loss values in NL descriptions than in the source code. The LLM also has a higher perplexity for NL descriptions than for source code. We show a few examples in Figure 2 to visualize the differences.

These differences arise because source code has a more strict syntax and semantics compared to NL. Therefore, when generating code, some tokens can be easily inferred based on grammatical rules, and LLMs can confidently generate these tokens with low loss values. In contrast, NL allows greater freedom in word usage and often presents multiple viable choices for the same context, which results in high loss values.

Additionally, we find that **a number of peaks occur in loss distributions of source code, i.e., tokens with a higher loss value than nearby tokens.** A higher loss value means that it is more difficult to make correct predictions at these locations. We call the tokens with peak loss values **challenging tokens**. As for the tokens that have low loss values, the model has more confidence in predicting them correctly. We refer to them as **confident tokens**.

### In-depth Study of Code Tokens

This section provides a detailed investigation of the challenging tokens and confident tokens. We analyze samples from the MBPP (Austin et al. 2021) (500 samples), HumanEval (Chen et al. 2021) (164 samples), and APPS (Hendrycks et al. 2021) (train set: 5000 samples) datasets. We use CodeGen-2B to generate the ground truth code in these datasets and collect the corresponding loss values.

First, we define the predictive difficulty (PD) of a token, which is the rank (%) of the token loss among all token loss values in the code snippet, and compute PDs for all tokens. Then, we split each code snippet into code lines and inte-

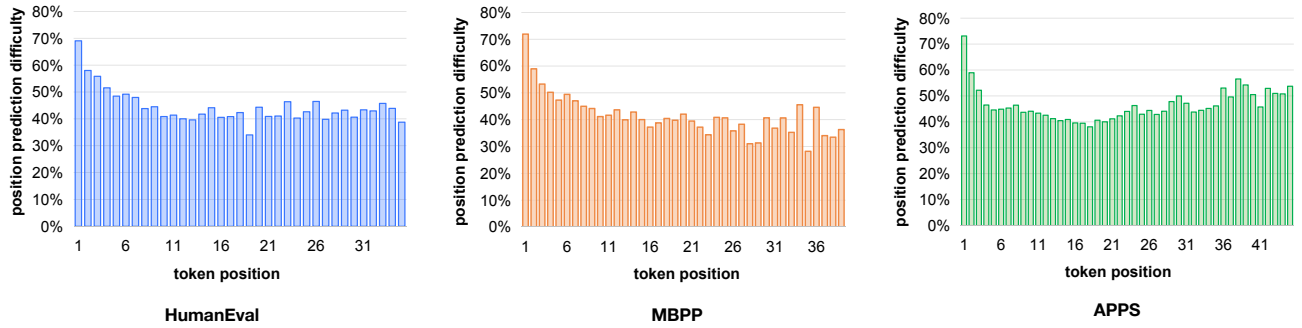


Figure 3: The prediction difficulty of different positions, the x-axis represents the position of the token within the line of code, and the y-axis represents the prediction difficulty of the token.

grate (average) the tokens' PDs in the same position of different code lines to obtain the position prediction difficulty. Finally, we statistically average the position prediction difficulty across all codes in the dataset. We omit positions where the cumulative data numbers of these positions are below 5% of the overall token numbers. Figure 3 shows the position prediction difficulty of different token positions. The results reveal the regularity that the tokens in the first position of code lines have the highest prediction difficulty of 69.0% (HumanEval), 73.1% (MBPP), and 71.9% (APPS). Therefore, we assume that the first position in each code line is where challenging tokens tend to appear and conduct further experiments.

We distinguish between challenging tokens and confident tokens based on PD. We introduce a threshold $H$ to separate two types of tokens. Tokens with PD $>$ $H$ are categorized as challenging tokens, while tokens with PD $<$ $H$ are categorized as confident tokens. We set $H$=0.9, and the proportion of challenging tokens appearing in the first position was 24.8% (HumanEval), 22.6% (MBPP), and 28.1% (APPS), respectively. The results confirm our assumption that the challenging tokens are not randomly distributed for different locations, they tend to appear at the first position of each code line.

To further investigate the properties of challenging tokens, we distinguish the tokens at the first position of each line based on whether it is the initial token of a code block. A code block in Python is a piece of Python program text that can be executed as a unit. The code block begins at the first indented statement and continues until the indentation returns to a previous level. The initial token of the code block has a prediction difficulty of 79.8% (HumanEval), 83.4% (MBPP), and 82.5% (APPS) which is significantly higher than the first positions of other code lines. We derive that among the first positions of code lines, **the initial token of each code block is the most likely place for a challenging token to appear**.

This can be attributed to the fact that LLMs need to determine the next control structure after a block of code is presented, which increases prediction difficulty. LLMs can easily generate the next token after receiving the initial token since the strict syntax rules limit the scope of variations.

## AdapT Sampling

In light of our findings, we propose a simple yet effective decoding method, AdapT sampling (Adaptive Temperature Sampling), which adjusts the temperature coefficient $T$ for different tokens. Specifically, for the challenging tokens, which LLMs struggle to predict correctly, AdapT sampling uses a high temperature coefficient which introduces more diverse tokens. On the other hand, for confident tokens, the temperature coefficient is set to a small value to minimize randomness noises. Specifically, we formulate AdapT sampling as follows:

$$p'(y_t|y_{<t},x)) = \frac{\exp(\frac{(u(y_t|y_{<t},x))}{T(t)})}{\sum_{j=1}^{n}\exp(\frac{(u(y_j|y_{<t},x))}{T(t)})} \tag{2}$$

$$T(t) = \begin{cases} a & \text{if } y_t \text{ is the code block initial token} \\ b & \text{else} \end{cases} \tag{3}$$

where $T$ is the temperature coefficient and $t$ is the sample timestep. $a, b \in [0, 1]$ $(a > b)$ are hyperparameters that control the degree of sampling randomness.

## Experiments

### Benchmarks

**HumanEval** (Chen et al. 2021) is a Python code generation benchmark with 164 test samples. Each sample consists of a manually written programming problem, which consists of a natural language requirement, a function signature, and several unit tests. It asks LLMs to generate the function body based on the requirement and the signature. The unit tests are used to check the correctness of generated functions.

**MBPP** (Austin et al. 2021) contains 500 programming problems collected from real-world communities. Solving these programming problems requires simple numeric manipulations and the basic usage of standard libraries. Each problem contains an English requirement, a Python function signature, and three test cases. We take the requirement and the function signature as input and leverage LLMs to generate the function body. Then, the generated code is evaluated using test cases.

| Metric | pass@5 | pass@10 | pass@15 |
|---|---|---|---|
| *CodeGen*: | | | |
| SP, $T$=0.2 | 29.2 | 31.3 | 32.3 |
| SP, $T$=0.4 | 33.4 | 37.9 | 40.8 |
| SP, $T$=0.6 | 33.1 | 38.0 | 40.8 |
| SP, $T$=0.8 | 32.7 | 39.7 | 43.9 |
| AdapT | **34.4** | **40.1** | **43.9** |
| *InCoder*: | | | |
| SP, $T$=0.2 | 22.8 | 25.9 | 27.4 |
| SP, $T$=0.4 | 25.0 | 29.8 | 32.3 |
| SP, $T$=0.6 | 24.5 | 30.0 | 32.9 |
| SP, $T$=0.8 | 22.3 | 28.6 | 32.9 |
| AdapT | **25.8** | **31.6** | **35.3** |
| *CodeGeeX*: | | | |
| SP, $T$=0.2 | 24.1 | 27.1 | 28.7 |
| SP, $T$=0.4 | 27.0 | 30.5 | 32.3 |
| SP, $T$=0.6 | 27.7 | 32.5 | 35.4 |
| SP, $T$=0.8 | 27.1 | 33.0 | 36.0 |
| AdapT | **29.4** | **36.3** | **40.9** |

Table 1: The performance (pass@5, 10, 15) for CodeGen-2B, InCoder-6B, and CodeGeeX-13B on the HumanEval dataset using AdapT sampling and SOTA (SP) method.

| Metric | pass@5 | pass@10 | pass@15 |
|---|---|---|---|
| *CodeGen*: | | | |
| SP, $T$=0.2 | 33.1 | 36.5 | 38.4 |
| SP, $T$=0.4 | 37.0 | 42.1 | 45.0 |
| SP, $T$=0.6 | 37.1 | 43.5 | 47.0 |
| SP, $T$=0.8 | 35.2 | 43.1 | 47.0 |
| AdapT | **37.2** | **44.4** | **48.2** |
| *InCoder*: | | | |
| SP, $T$=0.2 | 23.0 | 27.4 | 29.2 |
| SP, $T$=0.4 | 26.0 | 29.6 | 32.4 |
| SP, $T$=0.6 | 24.6 | 30.4 | 34.6 |
| SP, $T$=0.8 | 23.6 | 30.6 | 34.6 |
| AdapT | **26.8** | **32.9** | **36.8** |
| *CodeGeeX*: | | | |
| SP, $T$=0.2 | 20.7 | 23.3 | 22.4 |
| SP, $T$=0.4 | 23.7 | 28.4 | 31.0 |
| SP, $T$=0.6 | 24.8 | 31.2 | 34.8 |
| SP, $T$=0.8 | 25.4 | 31.2 | 35.6 |
| AdapT | **25.8** | **32.2** | **36.0** |

Table 2: The performance (pass@5, 10, 15) for CodeGen-2B, InCoder-6B, and CodeGeeX-13B on the MBPP dataset using AdapT sampling and SOTA (SP) method.

## Base Models

In this paper, we select three representative open-source LLMs as base models: CodeGen, InCoder, and CodeGeeX.

**CodeGen** (Nijkamp et al. 2022b) is a family of LLMs pre-trained on a large amount of NL texts and source code. It is trained with a 635GB code corpus and 1159GB English text data. In this paper, we select the CodeGen-mono with 2 billion parameters as the base model.

**InCoder** (Fried et al. 2023) is pre-trained with a large corpus of permissively licensed code (216GB). It can perform code generation and code infilling. In this paper, we use a version with 6.7 billion parameters, named InCoder-6B, for code generation.

**CodeGeeX** (Zheng et al. 2023) is a multilingual LLM with 13 billion parameters. It is pre-trained on a large corpus of more than 20 programming languages and achieves impressive performance on code generation.

Despite OpenAI models' impressive performance, we can not access the probability distribution with only API calls. Therefore, we ignore these models in this paper.

## Baselines

In this paper, we choose the SOTA approach mentioned in the background section as the baseline. For the sake of brevity, we refer to the baseline (reshaping distribution with temperature sampling) as SP in the following sections.

## Implementation Details

We run all of our experiments on 2 NVIDIA V100 GPUs with 32GB memory. For our experimental datasets, the maximum generated length is 500. All experiments are conducted in a zero-shot setting which means we directly feed the input requirement into LLMs without examples. Then we extract generated programs from the model's output.

## Metric

**Pass@$k$** Pass@$k$ (Chen et al. 2021) measures the functional correctness of the generated code by executing test cases. We use the unbiased version of pass@$k$, where $n \geq k$ samples are generated for each problem, and $c \leq n$ is the number of correct samples that pass test cases. Pass@$k$ is calculated as follows:

$$\text{pass@}k = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4}$$

## Main Results

We apply AdapT sampling to three base models on two code generation datasets. The performance is shown in Table 1

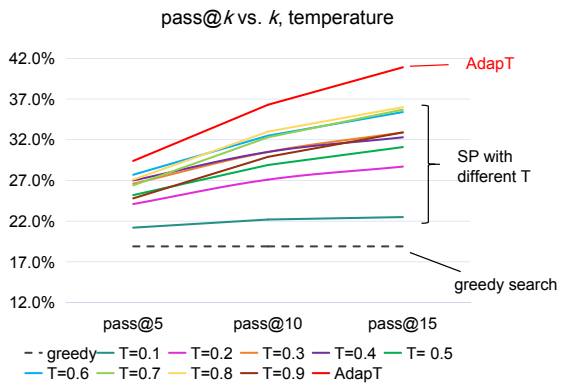


Figure 4: The performance of CodeGeeX-13B on the HumanEval dataset with dense temperature settings.

| Dataset | HumanEval | MBPP |
|---|---|---|
| *CodeGen*: | | |
| Greedy Search | 23.8 | 24.0 |
| SP-best | 22.2 | 24.6 |
| AdapT | 23.3 | 24.9 |
| *InCoder*: | | |
| Greedy Search | 14.0 | 12.3 |
| SP-best | 15.3 | 13.8 |
| AdapT | 15.2 | 14.8 |
| *CodeGeeX*: | | |
| Greedy Search | 18.9 | 13.6 |
| SP-best | 18.0 | 13.4 |
| AdapT | 18.8 | 13.5 |

Table 3: Pass@1 results for CodeGen-2B, InCoder-6B, and CodeGeeX-13B using different decoding strategies on HumanEval and MBPP datasets.

and Table 2. Following the previous works (Nijkamp et al. 2022a; Zheng et al. 2023; Fried et al. 2022), we set $p$ of top-$p$ sampling as 0.95 for all our experiments. Following the previous work (Chen et al. 2021), we set the values of temperature of baseline as 0.2, 0.4, 0.6, and 0.8 respectively. We experimentally select the values of $a$ and $b$. The sampling number $n$ is 15. As shown in Table 1 and Table 2, AdapT sampling outperforms the SOTA method in terms of pass@5, pass@10, and pass@15 on HumanEval and MBPP datasets. Pass@15 represents the number of problems solved since the sampling number $n$ is 15. AdapT sampling with the highest pass@15 indicates that it solves the most problems. Notably, on the HumanEval dataset, AdapT sampling can enhance the pass@15 of CodeGeeX from 36.0% to 40.9%, reaching a 13.6% improvement. Meanwhile, the number of solved problems increased from 60 to 67. On the HumanEval and MBPP datasets, CodeGeeX-13B can solve 14 and 36 previously unsolved problems with AdapT, respectively. When run with a constant value of $T$, increasing $T$ will improve the number of problems solved, but it may reduce pass@5 (indicates the proportion of correct answers sampled for each question) due to the introduction of randomness (Chen et al. 2021; Holtzman et al. 2019). On the other hand, AdapT sampling can dynamically adjust the sampling randomness, thus minimizing the noise associated with increasing $T$, hence consistently demonstrating an improvement in pass@5, 10, and 15.

## Analysis and Discussion

We conduct an in-depth and comprehensive analysis of AdapT sampling's capabilities.

**pass@1** The pass@1 metric represents the probability that, among multiple pieces of generated code, the first piece chosen would successfully pass a given test case. Note that the pass@1 are very strict metrics and are hard to improve. We compare the pass@1 results of AdapT sampling with greedy search (which usually has a high pass@1 value) and the best pass@1 results of SOTA (SP-best) and show the results in Table 3. Our method outperforms SP-best in 83.3% cases on the pass@1 metric. Meanwhile, AdapT sampling can reach a comparable pass@1 when compared with greedy search. Greedy search can only sample one answer per question, whereas our method can sample $n$ answers and increase the number of solved questions.

**Compare with Different $T$ Settings** To explore the upper bound performance of SP, we set the temperature value in SP more densely from 0 to 1, taking a value every 0.1 intervals. The results are shown in Figure 4. The performance is not displayed on the resulting graph when $T \geq 1$, since it drops significantly when $T \geq 1$. AdapT sampling significantly outperforms temperature sampling at all settings on pass@5, pass@10, and pass@15. The LLM can only answer 31 questions correctly with a greedy search. Using the AdapT sampling method, LLM can solve twice as many problems (67) as greedy search.

**Hyperparameters Analysis** There are two hyperparameters involved in AdapT sampling: $a$ and $b$. In this section, we examine how changing these two parameters affects the code generation results.

First, we fix $a = 0.8$, and we take a $b$ value every 0.1 step from 0.1 to 0.9, the experimental results are displayed in the upper part of Figure 5. AdapT sampling outperforms SP ($T = 0.8$) on pass@1, pass@5, pass@10, and pass@15 metrics when $b$ equals 0.3, 0.5, 0.6, and 0.7. Then we set $b = 0.3$, and vary the value of $a$ from 0.1 to 0.9 with a step of 0.1. The results are shown at the bottom half of Figure 5. It can be seen from the experimental results that when $a > b$, the results of pass@5, pass@10, and pass@15 can be effectively improved. Additionally, changing a from 0.2 to 0.9 can increase pass@15 from 32.9% to 37.8%, achieving a 14.9% improvement.

The results indicate that AdapT sampling can outperform SP under a variety of settings, which confirms the robustness of AdapT sampling over hyperparameters. The empirical hyperparameter guidelines are: for optimizing pass@$k$ ($k$ >1), set $a$ to approximately 0.8 and $b$ to around 0.5; for optimizing pass@1, set $a$ to approximately 0.2 and $b$ to around 0.01.

**Code Quality Evaluation** In this section, we analyze the quality of code generated by different sampling methods on the HumanEval dataset. We collect the execution results of the generated code of three models and present them in Figure 6. AdapT sampling can generate more correct samples (passed) than baselines on all models. Using AdapT sampling, CodeGen-2B, InCoder-6B, and CodeGeeX-13B can generate 517, 333, 382 correct codes, which outperforms SP up to 17.2%, 8.8%, 14.0% higher than sampling with the SP method, respectively.

There are fewer TypeError and SyntaxError in the code generated by the three models using AdapT sampling than SP. By using AdapT sampling in CodeGen and CodeGeeX models, the occurrence of NameError can be reduced by 44.3% and 23.4%. In CodeGen and CodeGeeX, AdapT sampling reduces the incidence of NameError by 44.3% and 23.4%. The else section in Figure 6 contains IndentationErrors, RecursionErrors, UnboundLocalErrors, RuntimeError, etc. These types of errors rarely occur in our AdapT

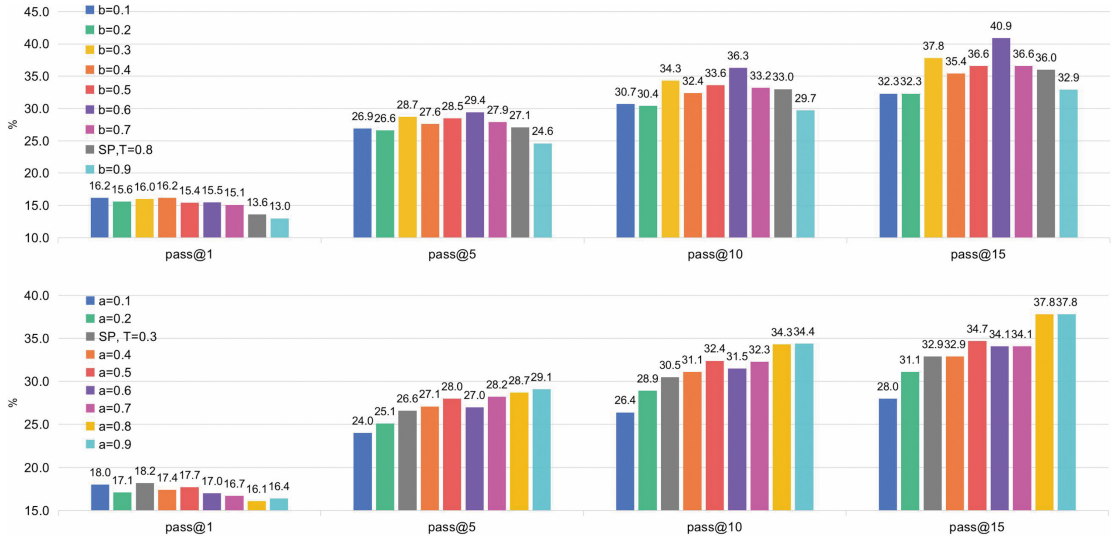


Figure 5: Quantitative analysis of the two hyperparameters ($a$, $b$) in AdapT sampling. The upper half shows the results vary b from 0.1 to 0.9 when fixing $a = 0.8$. The bottom half shows the results of fixing b = 0.3 and taking values of $a$ from 0.1 to 0.9. When $a > b$, the AdapT sampling can outperform SP with multiple settings. The model and dataset used in this section are CodeGeeX-13B and HumanEval.

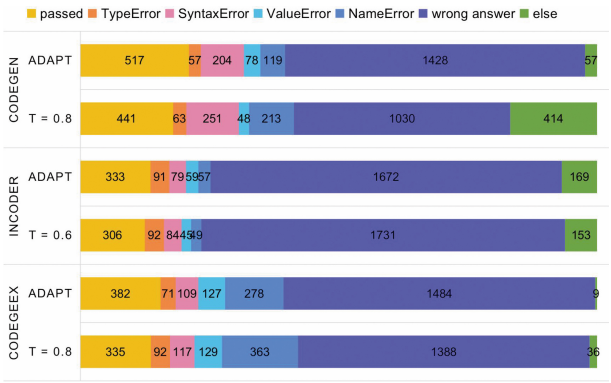


Figure 6: Evaluation of the generated code quality of AdapT sampling. Compared with the best-performed SP, AdapT sampling can reduce syntax errors and increase correct code sample numbers.

sampling. The reason for this is that AdapT sampling uses a smaller $T$ inside the code block, which improves the coherence of the sampled code, and therefore reduces syntax errors. For AdapT sampling, the most common error type is the wrong answer, showing that our method suffers from incorrect code logic. Taking steps to solve this issue can be a great improvement in the future.

## Future Work

In this section, we discuss the remaining challenges of decoding strategies in code generation and provide some possible directions to facilitate other researchers.

- We recognize some challenging tokens within statements, but their distributions do not show a clear statistical pattern. In the method designing process, we have experimented with various temperature tuning functions, such as linear decay function, exponential decay function, etc., without obtaining substantial improvement. In future work, we plan to use learning-based methods to adjust the temperature coefficients.
- In practice, software development often relies on specific domain knowledge, such as private code libraries and code specifications. Existing decoding strategies ignore these issues. In the future, we can introduce domain knowledge into the decoding process, improving the usability of code-generation LLMs in real-world scenarios.
- As shown in Figure 6, LLMs are suffering from incorrect code logic, and generating code from scratch is very challenging. In the future, we can design a multi-stage decoding strategy, which steers LLMs to generate code progressively. For example, LLMs first generate a natural language plan and then generate an executable program based on the plan.

## Conclusion

This paper is the first attempt at the LLM's decoding strategy for code generation. We statistically analyze the loss distribution of source code and find out that code tokens can be categorized into challenging tokens and confident tokens. Moreover, challenging tokens often appear in the initial of a code block. Based on the insights, we propose AdapT sampling which dynamically adjusts the temperature coefficient through sampling and has proven its effectiveness on code generation datasets. Finally, we present several challenges and insights in developing a more advanced decoding strategy for code generation and we look forward to further exploring its potential in future research.

## Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant Nos.62192731, 62072007, 62192733, 61832009, 62192730, and 62332012, the National Key R&D Program under Grant No.2023YFB4503801, and the Key Program of Hubei under Grant JD2023008.

## References

Ackley, D. H.; Hinton, G. E.; and Sejnowski, T. J. 1985. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1): 147–169.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program Synthesis with Large Language Models.

Black; and E, P. 2012. greedy algorithm, Dictionary of Algorithms and Data Structures. *US Nat. Inst. Std. & Tech Report*, 88: 95.

Bland, J. M.; and Altman, D. G. 1996. Measurement error. *BMJ: British medical journal*, 312(7047): 1654.

Brown, P. F.; Della Pietra, S. A.; Della Pietra, V. J.; Lai, J. C.; and Mercer, R. L. 1992. An estimate of an upper bound for the entropy of English. *Computational Linguistics*, 18(1): 31–40.

Brown, S. 2011. Measures of shape: Skewness and kurtosis.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language Models are Few-Shot Learners. *CoRR*, abs/2005.14165.

Brownlee, J. 2019. *Probability for machine learning: Discover how to harness uncertainty with Python*. Machine Learning Mastery.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Fan, A.; Lewis, M.; and Dauphin, Y. 2018. Hierarchical Neural Story Generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 889–898. Melbourne, Australia: Association for Computational Linguistics.

Freitag, M.; and Al-Onaizan, Y. 2017. Beam Search Strategies for Neural Machine Translation. In Luong, T.; Birch, A.; Neubig, G.; and Finch, A. M., eds., *Proceedings of the First Workshop on Neural Machine Translation, NMT@ACL 2017, Vancouver, Canada, August 4, 2017*, 56–60. Association for Computational Linguistics.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, S.; Zettlemoyer, L.; and Lewis, M. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, S.; Zettlemoyer, L.; and Lewis, M. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. In Vanschoren, J.; and Yeung, S., eds., *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Holtzman, A.; Buys, J.; Du, L.; Forbes, M.; and Choi, Y. 2019. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*.

Li, J.; Li, G.; Li, Z.; Jin, Z.; Hu, X.; Zhang, K.; and Fu, Z. 2023a. CodeEditor: Learning to Edit Source Code with Pre-Trained Models. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Li, J.; Li, G.; Tao, C.; Zhang, H.; Liu, F.; and Jin, Z. 2023b. Large Language Model-Aware In-Context Learning for Code Generation. *arXiv preprint arXiv:2310.09748*.

Li, J.; Li, Y.; Li, G.; and Jin, Z. 2023c. Structured Chain-of-Thought Prompting for Code Generation. *arXiv preprint arXiv:2305.06599*.

Li, J.; Li, Y.; Li, G.; Jin, Z.; Hao, Y.; and Hu, X. 2023d. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, 2124–2135. IEEE.

Li, J.; Zhao, Y.; Li, Y.; Li, G.; and Jin, Z. 2023e. AceCoder: Utilizing Existing Code to Enhance Code Generation. *arXiv preprint arXiv:2303.17780*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Mou, L.; Men, R.; Li, G.; Zhang, L.; and Jin, Z. 2015. On End-to-End Program Generation from User Intention by Deep Neural Networks. *CoRR*, abs/1510.07211.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022a. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022b. A Conversational Paradigm for Program Synthesis. *CoRR*, abs/2203.13474.

Runnenburg, J. T. 1978. Mean, median, mode. *Statistica Neerlandica*, 32(2): 73–79.

See, A.; Pappu, A.; Saxena, R.; Yerukola, A.; and Manning, C. D. 2019. Do Massively Pretrained Language Models Make Better Storytellers? In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, 843–861. Hong Kong, China: Association for Computational Linguistics.

Zhang, X.; Sun, M.; Liu, J.; and Li, X. 2021. Improving Diversity of Neural Text Generation via Inverse Probability Weighting. *CoRR*, abs/2103.07649.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; Su, T.; Yang, Z.; and Tang, J. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR*, abs/2303.17568.