Fine-tuning Large Language Models for Non-autoregressive Code Generation

Junjie Xu

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

Autoregressive generation involves a process of left-to-right token generation that mirrors a natural, human-like manner. It draws upon previously generated tokens to establish contextual cues and inter-token relationships. However, this step-by-step approach slows down inference due to its lack of parallelism. Conversely, Non-Autoregressive generation was conceived to expedite inference by generating all tokens simultaneously. Nonetheless, Non-Autoregressive models struggle to attain dependency within the sequence due to the difficulty of directly modelling the collective distribution of all tokens, leading to reduced accuracy compared to the autoregressive method. However, it's notable that programming languages display fewer left-to-right dependencies compared to natural languages. The structured and syntactical rules inherent in programming languages empower a token generation with reduced reliance on preceding tokens, making the programming languages easy to be predicted for Non-autoregressive Method. Additionally, some pre-trained large language encoder models don't adhere to the left-to-right autoregressive pattern, while the impact of these models and their potential for aiding non-autoregressive generation remains unexplored. Our project centres on fine-tuning pre-trained large language models to facilitate non-autoregressive token generation in code intelligence field. We also delve into exploring and evaluating conventional techniques commonly employed in this domain. Moreover, we leverage structured program representations to enhance the Non-Autoregressive model's ability to capture code sequence dependencies. Our experimentation reveals substantial improvements compared to prior methods in Non-autoregressive area. Furthermore, Our results indicate that our model performs nearly as well as the Autoregressive model when the output sequence length is restricted to 100 tokens, while achieving a notable decrease in inference time.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Junjie Xu*)

Acknowledgements

I want to express my heartfelt gratitude to my supervisor, Prarit Agarwal, Camille Tiennont, and Prof. Iain Murray for their tremendous help and guidance throughout this thesis. I'm also thankful to the School of Informatics at the University of Edinburgh for providing the necessary computational resources. Special thanks to my friends, Patrick Chen and Yijun Yang, for their helpful advice on my experiments.

Most importantly, I'm truly appreciative of the support and encouragement I received from my family members, Heping Xu, Yazhen Liao, and Yixin Tian.

Table of Contents

1 Introduction **1**

2 Background **5**

2.1 Seq2Seq Model . . . 5

2.2 Task Formulation . . . 7

2.3 Problem Statement . . . 7

2.4 Non-Autoregressive Decoding and Related Work . . . 9

2.5 Structured Program Representation . . . 10

3 Methodology **12**

3.1 Hidden Variable Modeling . . . 12

3.1.1 Statistical Method . . . 12

3.1.2 Deep Learning Method . . . 13

3.2 Guidance from AR model . . . 14

3.3 Hint from Structured Program Representation . . . 14

4 Experiment Settings **16**

4.1 Dataset . . . 16

4.2 pre-trained LLM . . . 17

4.3 Evaluation Metrics . . . 17

4.4 Model Settings . . . 18

5 Results and Analysis **20**

5.1 Baseline Method . . . 20

5.2 Hidden Variables Modeling . . . 23

5.3 Guidance from AR model . . . 26

5.4 Hint from Data Flow . . . 28

5.5 Trade-off Study . . . 29

5.6 Robustness Test 33

6 Conclusions **35**

6.1 Future Work 36

Bibliography **38**

A Comparative Examples of AR and NAR Model Outputs **43**

Chapter 1

Introduction

In recent years, there has been a notable upsurge in the interest and implementation of AI-driven tools aimed at simplifying code generation tasks. These tools have gained attention from both academia and industry, holding the potential to revolutionize software development by boosting efficiency and accuracy, and providing valuable support to developers in their work. Noteworthy examples of such tools include Github Copilot [12], Amazon CodeWhisperer [1], and OPEN-AI ChatGPT [31]. These tools showcase impressive capabilities in tasks like code suggestions, function completions, and code snippet generation. They are based on Neural sequence-to-sequence (seq2seq) models[2, 35, 38]. Given their impressive features and significant advantages, these AI-driven solutions have garnered widespread recognition and popularity among users. However, an essential aspect that profoundly impacts user satisfaction is the speed of these tools. Users prioritize responsiveness, as swift performance is crucial for seamless integration into developers' workflows and overall productivity. Sluggish or unresponsive behaviour can lead to frustration, disrupting the coding process and potentially discouraging users from adopting the tool.

In the field of seq2seq models and advanced deep learning architectures, token generation typically follows an autoregressive (AR) approach. This implies that each token prediction relies on the sequence of tokens generated before it. However, this autoregressive generation process inherently occurs sequentially, making it challenging to parallelize during inference. As a consequence, this approach leads to slower and computationally intensive performance [41]. The model must wait for the preceding token to be generated before it can predict the subsequent one, resulting in a linear increase in inference time as the length of the output sequences increases.

To expedite the inference process, a line of research has emerged, focusing on the

development of non-autoregressive(NAR) translation models. These innovative models aim to overcome the autoregressive dependency by adopting a different approach: they decompose the joint conditional probability and directly generate all tokens simultaneously. However, The loss of autoregressive dependency in non-autoregressive models can significantly impact the consistency of the output sentences, making the learning process more challenging and resulting in lower-quality translations. To address these limitations, previous research has primarily explored two main directions for improving non-autoregressive translation models:

1. Enhancing the expressiveness of the NAR model: One approach involves augmenting the NAR model with different components to improve the network structure's expressiveness. For instance, iteratively refining the target sentence generated by the NAR model [19, 11] or introducing additional modules to model hidden variables that capture dependencies between outputs [13, 28]. By introducing these additional components, the NAR model becomes more capable of generating high-quality translations that closely resemble those produced by AR models.

2. Training under the guidance of an AR model: Another direction of research focuses on training the NAR model with the assistance of an AR model. The AR model provides valuable information and guidance during the training process, helping the NAR model maintain coherence and context in its generated output. By leveraging the knowledge and structure of an AR model, the NAR model can learn to produce more consistent and accurate translations [21, 25, 16].

One notable aspect missing from the previously mentioned research directions is the exploration and utilization of the powerful pretraining capabilities of modern large language models (LLMs), such as BERT (Bidirectional Encoder Representations from Transformers) [7] and GPT (Generative Pre-trained Transformer) [4]. Also, none of them investiage the NAR generation on programming language that has less dependency need to be captured with the help of conditioning mechanism. In the current landscape of deep learning, transfer learning has emerged as a dominant paradigm, fine-tuning LLMs makes them capable of different downstream tasks including code generation. This project aims to explore the potential impact of pre-trained LLMs on the field of NAR methods, focusing on leveraging the knowledge stored within LLMs to improve generation speed while still benefiting from the comprehensive language understanding

captured during pretraining. We will primarily employ a Code-to-Code translation dataset to evaluate the performance of our NAR generation model. However, the ultimate goal is to make NAR decoding applicable to a wide range of tasks that require sequence generation. As such, we also intend to test our model's performance on code-to-natural language summarization tasks. The main contributions of this project are as follows:

1. Confirmation of Pretraining Benefits: Our experiments validate that leveraging pre-trained LLMs can be highly advantageous for NAR decoding. Furthermore, we establish that NAR models derive more substantial benefits from this approach compared to AR models.

2. Evaluation of Existing NAR Techniques: We systematically examine conventional methods employed in NAR models and assess their effectiveness after integrating pre-trained LLMs. Interestingly, we observe a reduction in the significance and improvement provided by these methods, owing to the mitigating effect of the pretraining process on the issues these methods aim to address.

3. Significant Enhancement via Data Flow: We investigate and test the impact of a specific structured program Representation, the data flow, on NAR decoding. The results highlight a substantial improvement in NAR decoding performance achieved by incorporating this data structure.

4. Trade-off Analysis: Our study includes a comprehensive analysis of the trade-offs between performance, speed, and the robustness of NAR models. We identify that when the speed factor is balanced, the performance of AR and NAR models becomes remarkably similar.

The main structure of this thesis is as follows. Chapter 2 offers an in-depth overview of the NAR and AR models, along with a detailed discussion of the task and problems associated with NAR models in the code intelligence field. Chapter 3 gives an examination of previous methodologies used in this field. It provides insights into the approaches and techniques employed in prior work. Chapter 4 outlines the experiment settings, including the parameters, configurations, and details about the evaluation datasets used in the project. In Chapter 5, the experimental results are presented, and a comprehensive analysis of these results is provided, offering insights into the performance and behaviour of the models. The final chapter concludes the project, summarizing the

findings and contributions. It also highlights the limitations of the study and proposes potential areas for future research.

Chapter 2

Background

This chapter aims to establish the necessary foundational knowledge for this project. In Section 2.1, we will present a concise overview of the Seq2Seq model, thereby setting the stage for comprehending the conventional deep learning method for sequence generation. Moving on, Section 2.2 will introduce the Task Formulation, providing a clear context for the problem being tackled. In Section 2.3, we will delineate the specific motivations, key issues, and project objectives. Additionally, Section 2.4 will delve into the existing landscape of Non-Autoregressive models, detailing various approaches that strive to overcome the autoregressive limitations and facilitate parallel token generation. Finally, section 2.5 provide an introduction to structured program representation and some related works that show its importance in code intelligence task.

2.1 Seq2Seq Model

Currently, two main types of Seq2Seq models are popular: the Encoder-Decoder model like BART, T5 [23, 34] and the Decoder-only model like GPT family, LLaMA[4, 33, 37]. While Decoder-only models tend to be larger in size, their computational resource requirements can be substantial. For the scope of this project, we will primarily focus on the more widely used Encoder-Decoder Architecture.

Modern deep learning models heavily rely on the Transformer architecture [39]. This architecture is widely used in various tasks, and in the case of encoder-decoder models, both the encoder and decoder components are based on the Transformer module as shown in Figure 2.1.

In the encoder part, the input sequence undergoes a series of transformations through the Transformer module and outputs the hidden states representation. This module

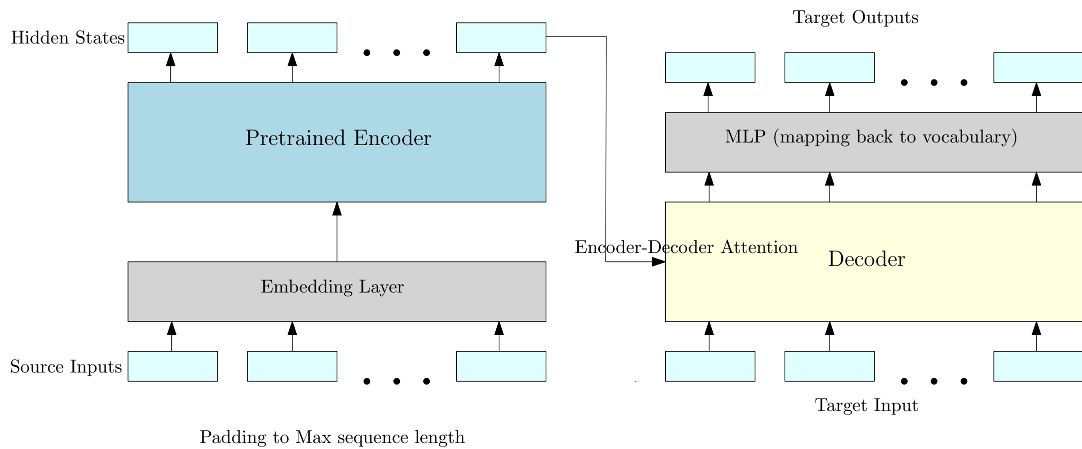

Figure 2.1: The overall structure of a seq2seq autoregressive encoder-deocder model, the decoder's input is target ids.

enables tokens in the sequence to attend to each other, allowing the model to gather information and comprehend the context of the entire sequence in both forward and backward directions. This bi-directional attention mechanism significantly enhances the model's ability to capture rich contextual dependencies. After obtaining the output from the encoder, the decoder component in the Transformer-based model takes the encoder's hidden state as input using the encoder-decoder attention mechanism and also conditions on the previous tokens to generate the outputs.

During the training process, the actual correct token from the previous time step is available, allowing for parallelization using masks to ensure that the model does not attend to future tokens. During inference time, when generating the output sequence, the AR model encounters a challenge due to the absence of actual correct former tokens. Since the model needs to wait for the preceding token before predicting the next one. This sequential generation process leads to slower inference. To find the most likely sequence, AR models also employ beam search [2], which explores multiple possible sequences in parallel. Beam search maintains a set of candidate sequences and expands them by predicting the next token for each candidate. This process continues until the maximum sequence length or a stopping condition is met. Thus, the actual inference time complexity is the sequence length (n) times the beam search size (b) (using the big O notation [5]: $\mathbf{O}(b \times n)$). Figure 2.2 is a visualisation of this comparison.

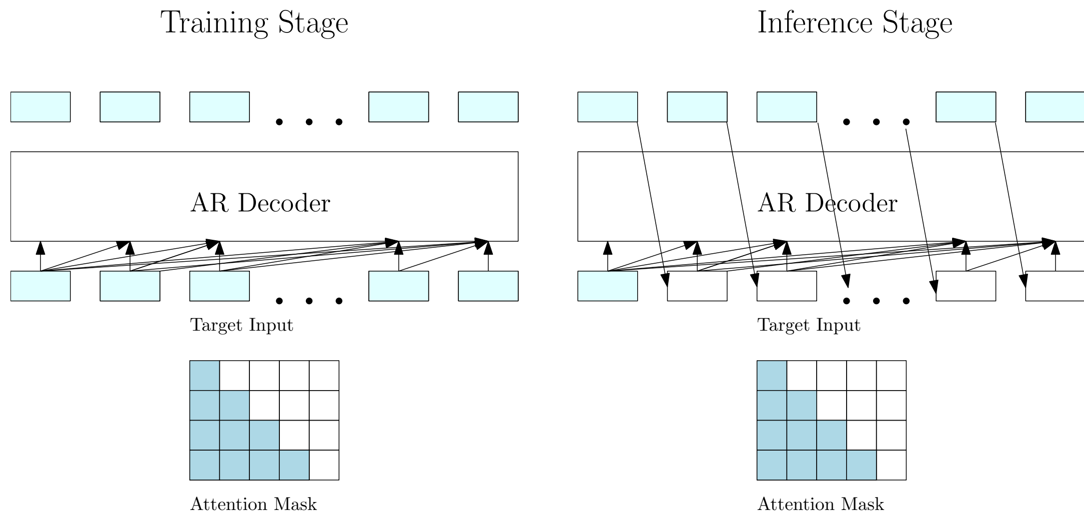

Figure 2.2: The decoder part of the Seq2Seq model during training and inference stage. The training stage use the exact target ids as input, the triangular attention mask make sure model does not attend the correct token at current position. During the inference stage, the predicted token from the previous time step will be used as input for the following steps. The First Token is the start of sentence token $< s >$, so is always available.

2.2 Task Formulation

It is a common belief that natural language exhibits strong dependencies, leading to the conventional use of autoregressive Seq2Seq models. These models generate the output sequence $\mathbf{Y} = y_1, ...y_{t'}$ by decoupling the joint distribution $P_\theta(\mathbf{Y}|\mathbf{X})$ (where $\mathbf{X}$ is the source sequence) into conditional probabilities $\prod_{t=0}^{t'} P_\theta(y_{t+1}|\mathbf{X}, y_t)$, where P_θ represents the learned distribution from deep learning structures. This approach effectively captures the dependencies between the output tokens, ensuring coherence and context in the generated sequence. In contrast, non-autoregressive methods predict each token y_t solely based on the input sequence $\mathbf{X}$ without considering the previously generated tokens. Therefore, non-autoregressive methods need to find a way to model the dependency in the output by utilizing the source input.

2.3 Problem Statement

This lack of capturing the inter-token dependency in the output sequence leads to the main problem that harms the performance of NAR known as the ”multimodality

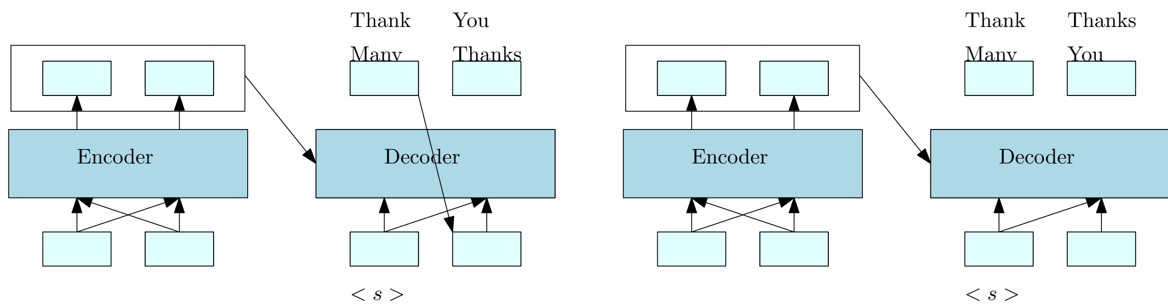

Figure 2.3: Examples of the ”multimodality problem” occur when there are multiple correct outputs for a given input, but the model struggles to choose between them. For instance, if the phrases ”Thank You” and ”Many Thanks” are equally likely in the dataset, the model might assign similar probabilities to tokens at each position. Without knowing the first token generated in the previous time step, the model may give equal probabilities to ”Thanks” and ”You,” leading to an incorrect translation like ”Thank Thanks” and ”Many You”.

problem” [42]. For instance, shown in Figure 2.3, consider the task of generating phrases that express gratitude, if phrases such as ”Thank you” and ”Many thanks”, both of which carry the same meaning and appear with close frequency in the dataset, due to the lack of inter-token dependencies, the NAR model may not correctly associate words to form coherent phrases. It will assign equal probabilities to the tokens in them at each position. This could lead to the generation of nonsensical and incoherent outputs, such as ”Many you” and ”Thanks thank,” where words from different phrases are mistakenly combined.

However, in the context of code generation, this challenge is somewhat less severe due to the inherent characteristics of code data. Unlike natural language, where word order and multiple synonyms introduce ambiguity, code data tends to have a smaller, constrained vocabulary. The code structure is more rigid, leading to fewer ways of expressing the same functionality. Consequently, the ”multimodality problem” is less pronounced in code generation tasks. The structured and concise nature of code allows NAR models to perform well. The limited vocabulary and strict syntactic rules enable accurate code sequence generation with reduced ambiguity.

An approach to mitigating the ”multimodality problem” in natural language tasks is using part-of-speech (POS) tags as additional inputs, as demonstrated by Yang et al. [42], POS tags can help narrow down the possible translations (even though there might be several correct options) by imposing constraints based on the tags, and also reducing the overall ambiguity in the sequence. In code generation, a similar effect

can be achieved through structured program representation such as Abstract Syntax Trees (AST), Data Flow and Intermediate representation(IR). These representations provide clear insights into the dependencies and structure of code sequences, mirroring the advantages of POS tags in natural language tasks. Furthermore, obtaining these representations is generally more straightforward than acquiring POS tags in natural language scenarios, primarily due to the inherently structured nature of code data that we previously discussed.

2.4 Non-Autoregressive Decoding and Related Work

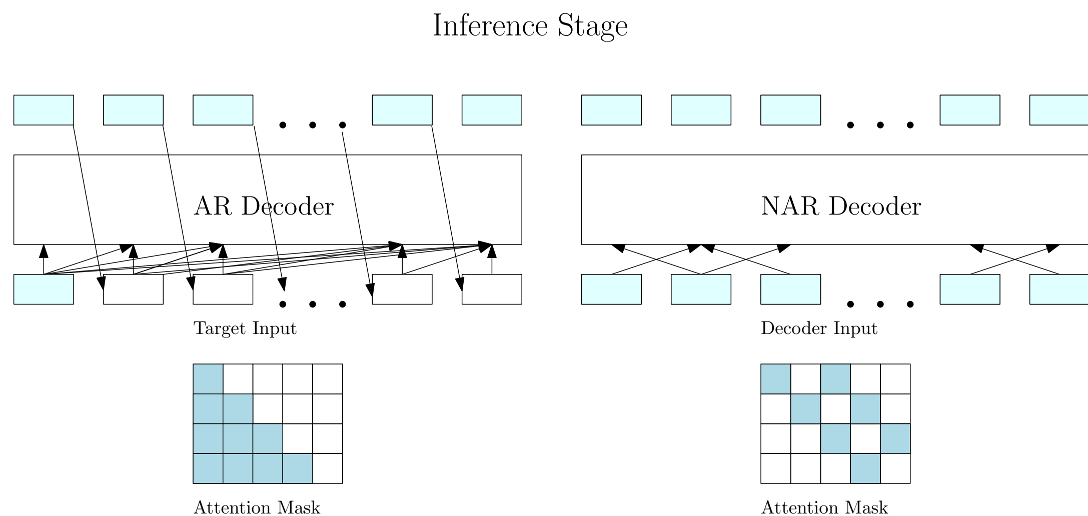

Figure 2.4: The figure shows a difference between the way AR and NAR models generate tokens during the inference stage. In the AR model, each token is generated step by step, using previous predictions as input and an attention mask. In the NAR model, there's no need for such iterative generation or attention mask. Instead, NAR models can limit token attention to a small ”window” around each token, which helps them work better without attending too much padding tokens. The example in the figure demonstrates a window size of 1, meaning each token only looks at its immediate neighbors.

The overall architecture of NAR and AR models bears similarity, but the key distinction lies in the decoder part as presented in Figure 2.4. In the AR model, the decoder relies on the previously generated token, following an autoregressive process, where each token is predicted sequentially. On the other hand, NAR models deviate from this sequential prediction approach. They attempt to decode all tokens simultaneously, without the need to wait for the previous token, resulting in the time complexity

at inference time being $\mathbf{O}(1)$. Thus, NAR models must adopt alternative decoding strategies instead of taking previous tokens as decoding input, as discussed in Chapter 1. Research on NAR models primarily focuses on two main directions: discovering more expressive representations for the decoding input and leveraging representations from autoregressive models.

In the first type of research direction, non-autoregressive models have explored innovative approaches to improve decoding. For instance, Gu et al. [13] proposed a method involving the modelling of hidden variables $\mathbf{Z}$ that effectively matches input source tokens to their corresponding target output tokens. By conditioning on these hidden variables, the model can generate tokens by copying input tokens, thereby reducing the dependency on sequential generation. Similarly, Ma et al. [28] introduced a different strategy by decoupling the conditional distribution $P_\theta(y_{t+1}|\mathbf{X}, y_t)$ into $P_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{Z})$. In this approach, an additional hidden variable $\mathbf{Z}$ is learned, which helps to reduce or reflect the dependency between output tokens. By incorporating $\mathbf{Z}$ into the decoding process, the model gains more flexibility and can generate tokens more efficiently, without strict sequential constraints. In addition, decode on the output of the NAR model itself is also available, Kasai et al. [19] and Ghazvininejad et al. [11] involve iteratively refining the target sentence generated by the NAR to improve the initial prediction which may not be perfect due to the challenges of non-autoregressive decoding and the multimodality problem. The second research direction aims to leverage the knowledge from AR models, often referred to as "teacher AG," to improve the Non-Autoregressive decoding process. It has been observed that the hidden states produced by AR models exhibit much smaller similarities with each other compared to NAR models. This difference in hidden state similarities is a significant factor contributing to the "multimodality problem" in NAR models. To address this issue, researchers have explored techniques such as sequence-level knowledge distillation [21]. Additionally, alignment between the AR and NAR models is used in conjunction with the attention mechanism to guide the NAR decoding process [25, 16].

2.5 Structured Program Representation

Numerous prior studies have highlighted the advantages of incorporating structured program representations such as Abstract Syntax Trees (AST), data flow, and Intermediate Representations (IR) into Language Model architectures for tasks related to code intelligence[3, 17, 20]. Those data are foundational components in computer

science and software engineering that provides structured, abstract information of various aspects of a program's structure, semantics, and flow. They encompass essential components used in the development and analysis of software, particularly in the context of compilers, interpreters, and static analysis tools. Structured Program Representation facilitates the transformation of human-readable source code into machine-executable instructions, enables optimization techniques, and aids in program understanding.

These structured program representations are particularly advantageous for NAR decoding. They provide valuable hints and guidance for modelling token dependencies and constraints, thereby helping to address the "multimodality" problem. For instance, when dealing with NAR models, the existence of multiple valid translation versions for a single input can pose challenges. However, the presence of specific data flow or AST structures serves as a constraint, limiting the translations to be consistent within that particular structure. This information serves as guidance to aid the NAR model in identifying the global optimum rather than getting stuck in local optima at each sequence position.

Chapter 3

Methodology

3.1 Hidden Variable Modeling

In non-autoregressive models, both the encoder and decoder parts employ the Transformer-based architecture. However, unlike autoregressive models, NAR models have the distinct characteristic of generating all words in parallel. This parallel generation process necessitates a different approach, as NAR models cannot use time-shifted target outputs or previously predicted outputs as inputs to the first decoder layer. As a result, there arises a need to identify a suitable representation that can serve as input for the decoder's functioning.

The decoder's input in NAR models is crucial for generating the initial output. It has been demonstrated that omitting inputs entirely to the first decoder layer or relying solely on positional embeddings can lead to significantly poor performance [13]. Without relevant inputs, the model lacks the necessary context and dependencies to guide the generation process effectively. Therefore, it is imperative to explore effective ways to model hidden variables that can enhance the decoding input to the NAR models. In this project, we aim to investigate two methods of finding better hidden variables that can serve as the decoder's input, we introduce them below.

3.1.1 Statistical Method

In this section, we build upon existing work that utilizes variational inference and training techniques. [28]. In addition to the standard encoder and decoder components, we introduce an additional part called the *posterior* part during the training process. The *posterior* part's role is to learn the distribution of $P_\Phi(\mathbf{Z}|\mu(\mathbf{X},\mathbf{Y}),\sigma^2(\mathbf{X},\mathbf{Y}))$ (μ and

σ^2 is the learnable mean and variance for the hidden variable Z) to model this additional hidden variable, z. which serves as the decoder's input for the NAR model. Since NAR models attend to all decoding inputs at each time step, it is crucial to ensure that the learned z does not simply copy the target tokens but captures the contextual interdependence among the tokens. To achieve this, we employ mask or token dropout strategies during training. These techniques help guide the learning process of z, preventing it from merely replicating the target tokens and encouraging it to grasp the underlying dependencies among the tokens [30, 7].

During inference time, we have no target sequence **Y** to feed to the *posterior* part, so we introduce another component, the *prior*, which directly models the distribution $P(\mathbf{Z}|\mathbf{X})$, it based on Glow [22], a flow-based architecture that ensuring the gradients remain tractable even for complex distributions so the standard back-propagation is feasible. Then the Kullback-Leibler (KL) divergence [6] between the *prior* and *posterior* distributions is employed to encourage the two distributions to be close to each other, allowing the *prior* to reconstruct the latent variable z without access to target input data. By incorporating the *prior* and *posterior* components in our framework, we leverage data to effectively decouple the dependency between the output sequence and the uncertainty in the generative process.

3.1.2 Deep Learning Method

The encoder part of modern large language models (LLMs) has become significantly large in size, offering substantial contextual understanding and language representation capabilities [7, 26, 14]. Unlike the autoregressive decoders that progress sequentially from left to right autoregressively, LLM encoders are bidirectional, allowing them to capture comprehensive contextual information from both directions in the input sequence. In this project, we propose that the hidden state output from a large pre-trained LLM contains valuable contextual information, which can potentially assist in modelling dependencies and improving the performance of non-autoregressive decoding. Given that the decoder will still require the hidden states of the source input via encoder-decoder attention mechanisms, too many modifications to the pretraining segment are ill-advised to prevent disruption of the acquired knowledge. Instead, we propose the integration of an additional deep learning module (such as extra transformer layers) to construct a proficient representation of the hidden variable, serving as input for the decoder. This module is similar to the aforementioned flow-based method. Moreover,

we can introduce hints or guidance by incorporating a loss function into the outputs from this module to tailor the process of learning the hidden variables.

3.2 Guidance from AR model

In the context of the NAR models, a pivotal factor that holds sway over their performance is the representation of the hidden state. Previous research indicates that hidden states learned in an AR manner tend to exhibit enhanced attention distribution and superior representation capabilities. These attributes provide essential hints to the hidden state, facilitating the capture of sequence dependencies [2]. To capitalize on this insight, researchers have explored methods to effectively imbue NAR models with guidance from AR models during the decoding process. One approach involves the incorporation of loss functions that encourage the alignment of the NAR model's hidden state representation with that of a Teacher AR model, alongside a loss function that enhances word alignment [25]. Given this, our project will also delve into the consequences of integrating guidance from a large pre-trained autoregressive language model. To be more precise, we intend to introduce supplementary components crafted to harness the hidden state representations garnered from an AR model. The ultimate objective is to reshape the NAR model's hidden state, aligning it more closely with the context and dependencies discerned by the AR model.

3.3 Hint from Structured Program Representation

Linguistic structure prediction has been a subject of extensive investigation in natural language processing. In previous works, researchers have explored various techniques to model and leverage syntactic structures (e.g. syntactic dependency trees and part-of-speech tags), primarily focusing on the decoder side [40, 8]. And it is also helpful for the parallel decoding process as it gives hints to capture word dependency [42]. In the domain of code generation, similar to the role of Part-of-Speech (POS) and syntactic dependency trees in natural language processing, code syntax information such as Abstract Syntax Trees (ASTs) and data flow play a crucial role in capturing the dependencies and structure of the code sequences. Numerous research efforts have demonstrated the significance of them in code generation tasks, and their adoption has been proven to be beneficial [15, 10].

We argue that since the NAR model can not exploit the dependency present in

the output tokens by conditioning on the previous tokens, the additional information provided by Abstract Syntax Trees (ASTs) and data flow becomes more beneficial to them. For instance, consider a simple example where the AST or data flow provides information about the parent node and its children. In such cases, the model should learn that the dependency between the children of the same parent node should be lower or nonexistent.

Chapter 4

Experiment Settings

4.1 Dataset

In our study, our main emphasis will be on evaluating the NAR model's performance in code-to-code translation tasks using various methods. Additionally, we will explore the model's generalization capabilities by applying it to code-to-natural language summarization tasks. This broader testing will help us understand how well the model can adapt to different types of tasks beyond its primary focus. Below are descriptions of the two datasets:

1. Code-to-Code Translation: In this task, we will explore the capability of our model to translate code from one programming language to another. The dataset consists of parallel functions within both the Java and C # versions of the codes that come from several public repos collected by CodeXGLUE [27]. The dataset has 10,300 training examples, 500 validation examples and 1000 test examples, the examples that cannot be parsed into an abstract syntax tree are removed, so we make sure we can get the structure program representation to test our hypothesis.

2. Code-to-Natural Language Summarization: In this task, we will investigate how our model can generate human-readable natural language summaries for given code snippets. We utilize Python data from CodeSearchNet [18], the examples that cannot be parsed into an abstract syntax tree (AST) and the token lengths that are either too long or too short are filtered out, resulting in 251,820 training examples, 13,914 validation examples, and 14,918 test examples.

The dataset we have chosen is commonly employed in the domain of code intelligence [15, 14, 24].

4.2 pre-trained LLM

To ensure meaningful and effective fine-tuning, it's essential that our pre-trained Large Language Model (LLM) is exposed to a substantial volume of code language data during its pretraining phase, particularly in the programming language of the dataset we have selected. For instance, if our dataset consists of Python code, it's crucial that the pre-trained model has undergone pretraining on a significant amount of Python language data. For our research, we leverage the encoder component of CodeBert [9], which encompasses 12 transformer layers. CodeBert has been pre-trained in a diverse range of programming languages, including Python, Java, JavaScript, PHP, Ruby, and Go. This diversity in pretraining makes CodeBert suitable for our chosen dataset. To maintain consistency and coherence in our architecture, we align our decoder with 6 transformer layers, following the design principles of CodeBert.

4.3 Evaluation Metrics

To evaluate our model's performance, we employ the bilingual evaluation understudy (BLEU) metric [32]. BLEU is commonly used in natural language processing tasks, including code intelligence, to assess the quality of generated text compared to a reference or ground truth. It consists of n-gram precision for each n-gram up to a specified maximum value (typically up to 4-gram), and also the brevity penalty (BP) that penalizes short translations by reducing the BLEU score:

$$\begin{aligned} BLUE &= BP * exp(\sum w_n * log(P_n)) \\ where\ \ BP &= min(1, (L_{generated})/(L_{reference})) \end{aligned} \tag{4.1}$$

w_n are the weights assigned to different n-grams, and P_n is the precision for n-grams.

For both tasks, we present the smoothed BLEU-4 score as our primary evaluation metric. Additionally, in the context of the code-to-code translation task, we complement the evaluation with an additional metric called CodeBlue [29]. CodeBlue is a specialized metric explicitly designed to assess the effectiveness of code translations, offering a more detailed evaluation that is tailored to programming language contexts. It combines several components, including the n-gram match, weighted n-gram match, syntax match, and dataflow match, resulting in a more fine-grained analysis of the translation effectiveness in a programming context.

4.4 Model Settings

To set the hyperparameters for our models, we thoroughly examined and considered the choices made in previous studies. Through testing on the validation set using the AR model, we identified the optimal hyperparameters that yield the best performance.

The hyperparameter settings presented in Table 4.1 serve as a foundational configuration for all models. However, for specific models that incorporate additional components such as ”posterior” or employ alternative methods, we provide clear explanations of the parameter modifications made. Any changes to the hyperparameters for such specific models are explicitly detailed.

Category	Hyperparameter	
architecture	# encoder transformers layers	12
	# decoder transformers layers	6
	max sequence length	300
	beam size	3
Optimization	Learning rate	5.00E-05
	weight decay	1.00E-03
	Learning rate schedule	AdamW
	# training steps	100000
	gradient clip	1
	Dropout rate	0.2
	Training Batch Size	32
	Evaluation Batch Size	8

Table 4.1: hyperparameters setting for the experiments

As the NAR model demonstrates a more significant speed advantage when dealing with longer sequences, our primary focus will be on the code-code translation task, which involves longer target sequences in the dataset. And we use the code-nl dataset to further test the robustness and adaptability of our model when the difference between source and target is large.

For all the additional modules incorporated into the encoder-decoder architecture, we ensure that each module is a 4-layer Transformer or with equivalent parameters. Additionally, we opt to freeze the lowest 4 layers of the encoder in these models. This approach aims to prevent excessive modification of the pretraining model’s acquired understanding in the process of modelling hidden variables. By maintaining an equal

number of learnable parameters, we also ensure a more fair and meaningful comparison between the different models.

In our NAR model, we employ an attention mask with a window size of 2. This design enables tokens at each position to attend to their two neighbouring tokens. Additionally, the CLS token (i.e. The first token) is made attendable at all positions. Including the CLS token in all time steps is beneficial due to its role as an overarching representation of the sentence's meaning. This inclusion aids in predicting tokens across all positions and offers guidance for aligning the source and target sequences within the sequence-to-sequence model.

Chapter 5

Results and Analysis

Analyzing NAR and AR models inevitably involves a trade-off between speed and performance. Initially, we will establish a baseline model and assess the impact of pretraining Language Model Models (LLMs) on NAR models' performance. By setting this initial benchmark, we can analyze how various model enhancements and optimizations impact performance while keeping computational speed as a critical constraint. our experimental approach follows the routine outlined in chapter 3, including investigating the effects of incorporating hidden variables, the guidance from the AR model and structured program representation. Once we identify the most effective model achieved through the methodology discussed in Chapter 3, our next step will be to compare it with the AR model. In this comparative analysis, we will examine the trade-off between the two models, considering both their performance and computational efficiency.

5.1 Baseline Method

As discussed in the preceding chapter, the NAR model can iteratively build the input by incorporating each output, it needs all decoder's input at the beginning for its parallel decoding. A simple way in previous machine translation studies is to "Copy" source input as the decoder input [13]. This method can encounter situations where the output translation at a specific position is not inherently linked to the corresponding input position. This can result in non-trivial alignments, as the translation text's order may not necessarily mirror that of the source input. Moreover, instances might arise where a single token in the source reflects multiple tokens in the translation, or conversely, multiple tokens are condensed into one. These scenarios can introduce intricate challenges in preserving positional order and alignment between the translated sequence and the

source, further exacerbating the complexity of the decoding process using the source inputs.

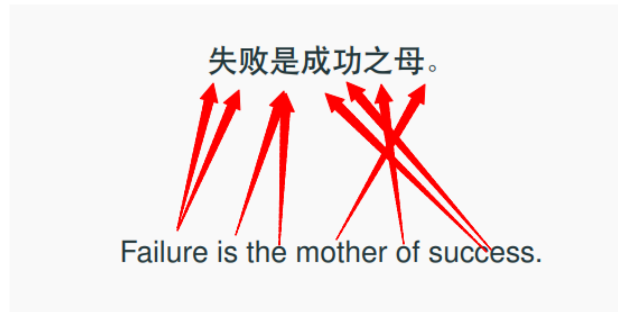

Figure 5.1: Example of the alignment problems between the source input ids and the generated translation. We can observe that the token ”failure” in the source input leads to two tokens in the Chinese translation, while ”is the” corresponds to only one token. Additionally, the order of tokens changes in the translation text. This phenomenon demonstrates the complexities and challenges in maintaining consistent alignments and dependencies between source and target tokens, particularly in the context of NAR decoding that can not use target ids as input.

Thus, instead of simply copying source inputs uniformly, previous work can investigate the effect of modelling additional hidden variables that indicate whether to copy source tokens multiple times or change their order. Here for the baseline method, we keep it simple by not introducing such hidden variables. The decoder inputs' length is the max sequence length we set and it learns to generate the EOS (end of sentence) token to indicate the output should stop there.

We also introduced a modification that entails decoding not directly from the source input but from the hidden states of the source input. The idea is that the similarity between the hidden states of the source and target is higher than the source and target inputs before encoding, so the hidden states may have more information that is helpful to the decoding process and we leverage the benefits of pretraining in this way.

In our initial experiments, we aim to assess the utility of a pre-trained LLMs encoder for NAR models. Since the pretraining process endeavours to capture contextual meanings of the input, the encoder encodes inputs with similar meanings to be close to each other, those captured information can be helpful for generating targets. Consequently, the most straightforward representation for decoding input is the hidden state of the source input. Due to the limited number of studies incorporating LLMs into the NAR field, and the absence of such research specifically in the realm of code intelligence

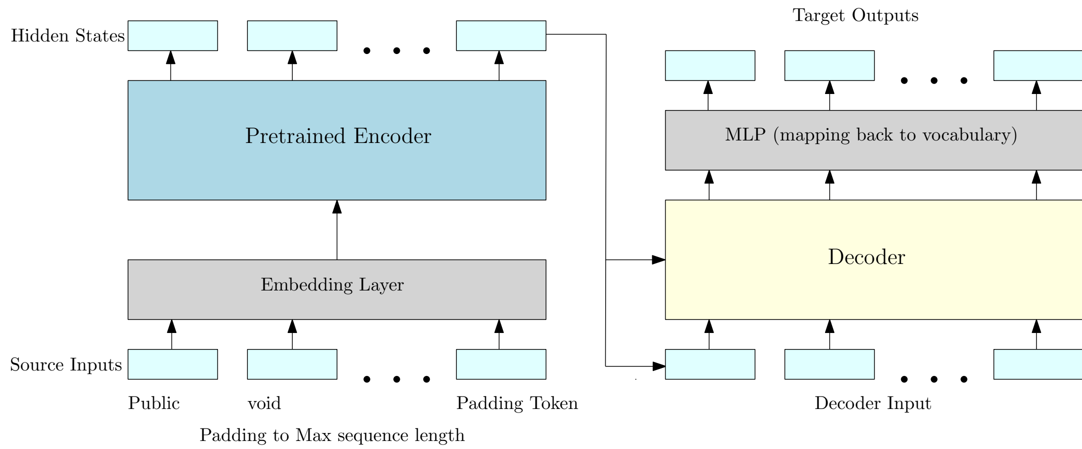

Figure 5.2: The baseline NAR model, where the decoder input consists of the hidden states from the source input.

tasks, We will adopt this configuration as the baseline for our project and assess the performance improvements we can achieve while operating within its speed constraints. We compare it with a model that without any pretraining to see pre-trained LLMs is helpful to NAR models. we ensure that the size and structure of the two models are the same, the model's structure is shown in the Figure 5.2.

It should be noticed that when we set the decoder input length to match the maximum sequence length, there's a possibility of having numerous padding tokens within each input sequence. In the AR method, it employs a mask to mitigate the impact by only allow each output to attend only to the former decoder's input. In our case, we restrict each input token to attend only to neighbouring tokens within a specified range during the generation. This enables us to avoid excessive attention to padding tokens. Interestingly, we find that the length of the neighbourhood within which each token can attend doesn't significantly influence performance (even if no mask is applied, i.e. attending to the padding tokens, does not hurt the performance too much).

NAR Model	BLEU-4
fully-random initialized	1.84(±0.45)
baseline(decode on source hidden state)	49.06(±0.88)

Table 5.1: Performance comparison between fully random initialized model and NAR model that decoder on source hidden state.

Through our investigation shown in Table 5.1, we observed a substantial difference

in performance when comparing the adoption of pre-trained encoders and randomly initialized transformers in the NAR model, in contrast to similar comparison experiments conducted in the AR model [27]. Our findings suggest that in the NAR model, the task involves learning how to capture and disentangle dependencies within the input sequence, which is distinct from the AR model's ability to condition on previous tokens. As a result, the complexity of the NAR task is considerably higher, making pretraining significantly more crucial for enhancing the model's performance. Also, another factor is that the LLM encoder also be pre-trained in a way that predicts tokens using the information from all the other tokens rather than autoregressively left-to-right, resulting in the hidden states from it is more beneficial for NAR methods.

5.2 Hidden Variables Modeling

We first try the method of adopting a flow-based statistical inference approach to model hidden variables for decoding input. In comparison to the baseline model, this approach introduces a *posterior* module, which is designed to model hidden variables suitable for decoding input while effectively capturing the dependencies and alignments discussed earlier. During training, both the target and source are provided to the *posterior* module, enabling it to learn and refine the hidden variables' representation. However, during the inference stage, the target input is not available. Despite this absence, we optimize the *prior* distribution using the KL-divergence loss between it and the *posterior*, thereby ensuring that it can generate the most accurate hidden variables even in the absence of target information.

We set the *posterior* to be a 4-layers transformer architecture and the flow-based *prior* module is similar to the previous work on Neural Machine Translation task [28]. The model's structure is illustrated in Figure 5.3. Through the experiments, we encountered challenges during the training process. Notably, two losses conflicted, impacting the model's convergence and performance. The first loss, the reconstruction loss, aimed to make the hidden variable with sufficient complexity to effectively convey meaning for improved decoding outcomes. On the other hand, the second loss, the KL-divergence loss, aimed to simplify the distribution of hidden variables, enabling the *prior* distribution to approximate the *posterior* distribution.

Although this trend has also been shown in the original paper. We further notice that introducing the pre-trained model leads the reconstruction error to be significantly lower and the variance of the *posterior* becomes close to 0, making the KL loss to be

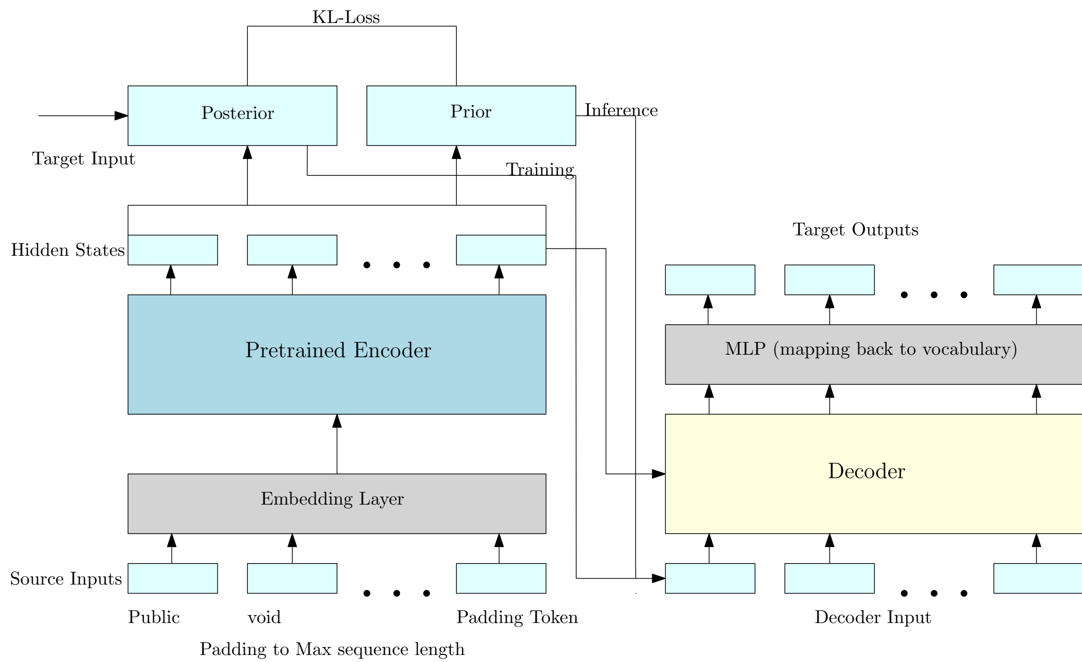

Figure 5.3: The structure of the model that utilizes a Flow-Based architecture for Hidden Variable Modeling, where the hidden variables serve as inputs to the decoder.

high and hard to converge. so even the hidden variable from *posterior* can result in a very good outcome, during the inference stage the performer is not comparable. and if we give more weight to the KL-loss to make sure the *prior* can be close to the *posterior*, the reconstruction loss will increase to the level as there is no pretraining encoder and can not converge like before.

Hence, our investigation led us to a realization that the outcomes we could attain appeared to be close to the original paper's results, where pretraining was not employed (adding a pre-trained encoder dose not help). Interestingly, introducing a pre-trained encoder within this framework seemed to amplify training complexities, resulting in a distribution that struggled to effectively model the hidden variable. This complexity suggests that a more expansive model capacity might be necessary to accommodate the nuances introduced by the pre-trained encoder.

The statistical inference approach involving *posterior* and *prior* modelling did not yield satisfactory results. In response, we explored alternative solutions within the conventional deep learning framework. Our approach involved incorporating a deep learning hidden variable predictor module, serving as a predictor for modelling the hidden variable both during training and inference, thereby it can not take the target as the input and learn the dependencies and alignments in the targets as the posterior module does. Instead, it needs other information to guide its behaviour. We tested that

if we can use the target hidden states as the decoder's inputs, we can get state-of-art performance like AR models do, this means the hidden states presentation of pre-trained encoder contains valuable information for non-autoregressive decoding. And we further notice the pretraining will lead the hidden state of the source and target to share a high similarity. Thus, To emulate the approach seen in the earlier statistical method, we introduced a cosine similarity loss between the hidden variable predictor's output and the target hidden state. This strategic addition aims to facilitate learning of capturing the intricate relationships required for accurate decoding, and also we believe this can help to address non-trivial alignments and one-to-many and many-to-one matching problems we mentioned above.

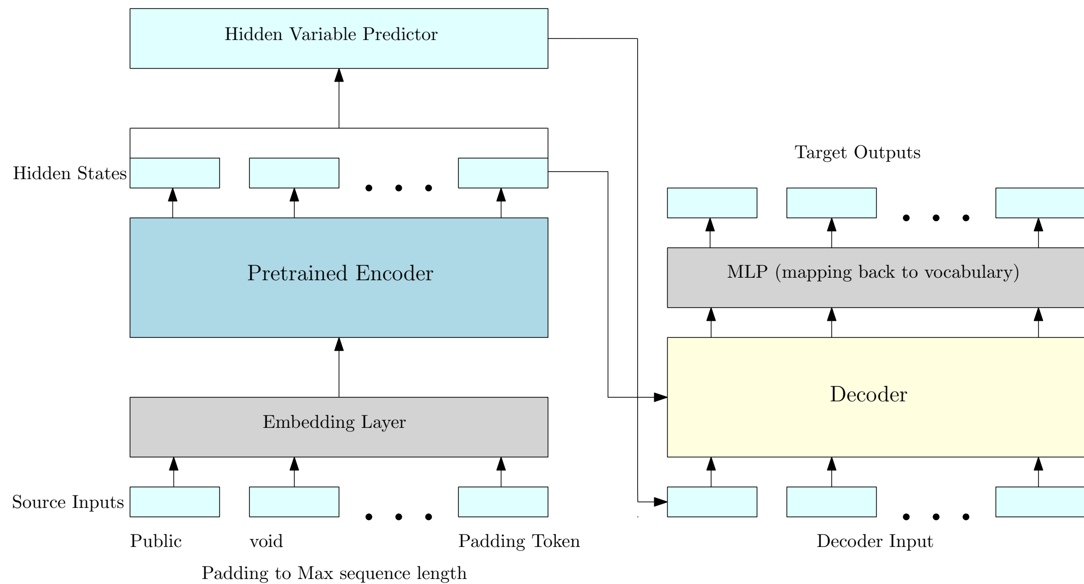

Figure 5.4: The structure of the model that utilizes a deep learning architecture for Hidden Variable Modeling, where the hidden variables serve as inputs to the decoder.

NAR Model	BLEU-4
baseline(decode on source hidden state)	49.06($\pm$0.88)
baseline+hidden variable(flow-based)	9.12($\pm$2.54)
baseline+hidden variable(dl-based)	50.62($\pm$0.57)

Table 5.2: Performance comparison between baseline model and models that utilize the additional hidden variables.

Table 5.2 reveals that the pre-trained encoder is already proficient in providing an optimal representation for non-autoregressive decoding. Interestingly, the introduction

of an additional flow-based variational component, which has shown utility in certain contexts, proves to be ineffective in our case. Furthermore, the additional guidance derived from the target hidden state offers only little improvement. This suggests that the challenges outlined earlier regarding non-trivial alignments and complex matching problems do not significantly impact performance.

5.3 Guidance from AR model

As stated in previous work, the guidance from the hidden state output of an AR model can help to improve the performance of NAR generation [25]. Research has shown that when the hidden states exhibit higher similarity, there's an increased likelihood of encountering the "multimodality problem", and generally, the similarity in AR model hidden states is notably lower compared to that observed in NAR models.

By leveraging the representations from a pre-trained encoder, we expect that the similarity among the hidden state outputs might not exhibit a high degree of resemblance, as would be the case in a fully randomly initialized NAR model. To validate this hypothesis, we conducted a comparison between it and our best model above (hidden variable(dl-based)) using only several input examples. The computed average similarity for the randomly initialized NAR model was found to be approximately 0.9, whereas, for our improved NAR model, the average similarity was around 0.6, but it remains significantly lower than what is typically observed in an AR model, where previous studies have indicated that approximately 95% of the similarity values between hidden states fall below the threshold of 0.5.

Thus we adopt their method to utilize the hidden state from the AR model (use hint from AR model) for getting better hidden states representations by introducing implicit loss:

$$
\begin{gathered}
L = \frac{2}{(T_y - 1)T_y N} \sum_{s=1}^{T_y - 1} \sum_{t=s+1}^{T_y} \sum_{l=1}^{N} \phi(d_{st}, d_{tr}) \\
where \;\; \phi(d_{st}, d_{tr}) = -\log(1 - d_{st})
\end{gathered} \tag{5.1}
$$

where d_{st} is the cosine similarity within NAR model, and d_{tr} is the cosine similarity within AR model. Also, it has additional hyperparameters to control the penalty.

The idea behind this approach is that straightforward regression such as regularising the L_1 or L_2 loss on each hidden states pair has been shown will impede the learning

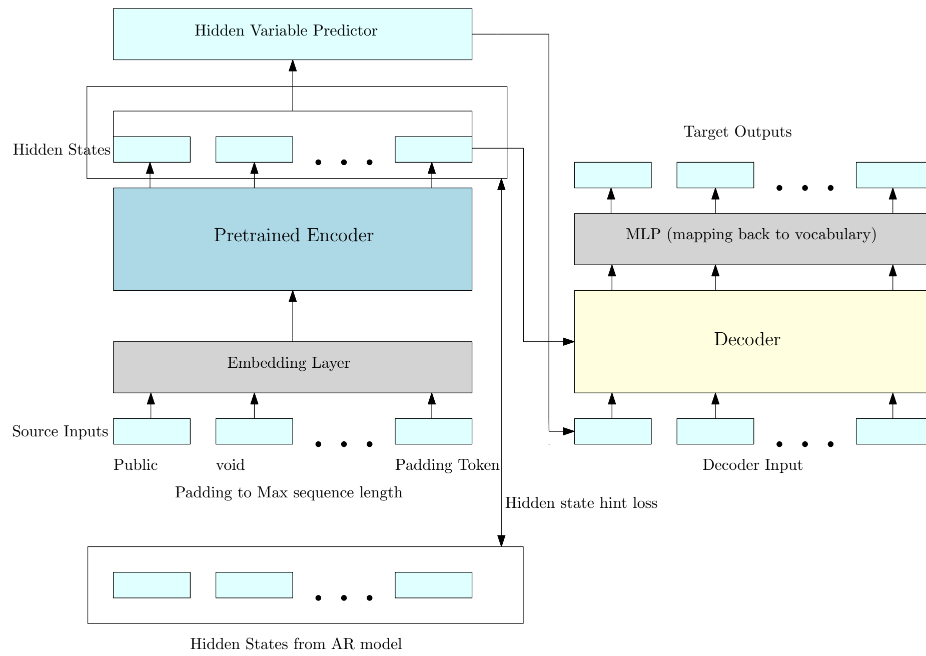

Figure 5.5: The structure of the model that employs a deep learning architecture for Hidden Variable Modeling, where the hidden variables are employed as inputs to the decoder. Additionally, the model incorporates hints from an AR model to refine the hidden state representation.

process and lead to unsuccessful outcomes.

We maintained the structure and model size, then adopt the loss equation 5.1 between this AR model's hidden state and the NAR model's to guide the NAR model's behaviour. The model's structure is illustrated in Figure 5.5. We find the average similarity between the hidden states of our NAR model decreases to 0.5, around 55% of the similarity values between hidden states fall below the threshold of 0.5 after using the guidance from the AR model.

NAR Model	BLEU-4
baseline(decode on source hidden state)	49.06(±0.88)
baseline+hidden variable(dl-based)	50.62(±0.57)
baseline+hidden variable(dl-based)+ hint from AR	52.32(±0.39)

Table 5.3: Performance comparison among the baseline model, the top-performing model mentioned earlier, and the model that incorporates hints from an AR model.

Our experimental results indicate that our model has achieved an approximate

4% improvement in performance, this improvement is notably smaller than the 30% enhancement observed in the original paper (increasing from 17.69 to 25.20 on the WMT14 dataset). We believe this is due to the utilization of a pre-trained encoder in our approach. The pre-trained encoder enables our model to benefit from high-quality hidden states, which may already possess a satisfactory level of quality. Furthermore, the baseline model we employed initially demonstrated commendable performance. Consequently, incorporating hints from the AR model to modify the hidden states appears to yield a relatively less significant impact, given the strong foundation established by the pretraining and the baseline performance.

5.4 Hint from Data Flow

Extensive research into LLMs has consistently highlighted the benefits of integrating data flow or Abstract Syntax Trees (AST) into the pretraining phase. This incorporation has been shown to yield noticeable performance enhancements in various code intelligence tasks [15, 36]. By leveraging the structural information provided by data flow or AST, pretraining models can gain a deeper understanding of the underlying code structures and the relationships between different program elements.

The utilization of data flow or AST as supplementary contextual information empowers the language model to capture intricate programming language nuances, help to model the dependency between the tokens and improve NAR generation capabilities.

The CodeBleu serves as a valuable tool that goes beyond measuring simple n-gram matches in code generation. It provides a comprehensive assessment by evaluating not only n-gram matches but also syntax and dataflow alignment. We analyse the CodeBlue of our best model above and find the data flow match score is much lower compared to an AR model. This outcome aligns with our initial expectations. Data flow inherently captures the data dependencies, indicating which variables originate from specific sources. Given that the NAR model lacks the condition mechanism present in the AR model, it becomes challenging for the NAR model to effectively capture these intricate data dependencies.

To maintain fairness in our comparisons, we chose not to switch the base encoder to one that has pre-trained on data flow, even though we anticipated enhanced performance if such a switch were made. Instead, we use a simpler approach by integrating data flow directly into the input and encoding it along with the source.

Table 5.4 and Table 5.5 demonstrate a significant benefit in incorporating data flow

NAR Model	BLEU-4
baseline(decode on source hidden state)	49.06($\pm$0.88)
+hidden variable(dl-based)+ hint from AR	52.32($\pm$0.39)
+hidden variable(dl-based)+ hint from AR and DF	60.31($\pm$1.58)

Table 5.4: Performance comparison among the baseline model, the top-performing model mentioned earlier, and the model that incorporates hints from data flow.

CODEBLEU	ngram	weighted ngram	syntax_match	DF_match
before adding DF	49.59	51.69	55.73	37.35
adding DF	64.04	65.09	68.26	61.44

Table 5.5: Detailed CodeBleu comparison between the model before adding hint data flow and after including this enhancement.

information into the model. This augmentation notably enhances the model's capacity to capture the intricate dependencies and structural aspects present in programming data, leading to a significant improvement in the data flow match metric. Consequently, this enhancement significantly reduces the disparity between the data flow match metric and other evaluated metrics. By integrating data flow information, the model achieves a better understanding of how data elements interact and traverse within the program. This enriched representation directly contributes to a better alignment of the generated code with the desired outcomes, leading to a noteworthy enhancement in performance across all the evaluated metrics.

These findings underscore the pivotal role of data flow information as a valuable contextual feature in the NAR generating process. Its incorporation substantially narrows the gap between data flow match and other performance measures, ultimately elevating the overall efficacy of the model in code intelligence tasks and further highlighting the importance of domain-specific structures in driving performance improvements.

5.5 Trade-off Study

As previously discussed, a trade-off invariably exists between performance and speed. The time complexity of AR is $O(b)$, while that of NAR is a more efficient $O(1)$. Considering this, if we maintain a consistent beam size, the NAR model yields time savings

proportional to the length of the generated target sequence. Longer target sequences considerably benefit from the NAR model's constant time complexity, contributing to more efficient processing and faster results compared to the AR model.

On the other hand, longer sequences present a challenge in capturing their inherent dependencies, they are more likely to contain intricate structural complexities and long-term dependencies, which can lead to difficulties for the NAR model. This observation is clearly illustrated in Table 5.6 below, the model performs better on short target sequences. We also provide a real-time measurement example in Table 5.7 to offer a concrete understanding of the inference speed comparison between the NAR and AR models.

Sequence length	NAR model	Time complexity	AR model	Time complexity
300	60.31	O(1)	76.42	O(3×300)
150	72.91	O(1)	76.49	O(3×150)
100	75.29	O(1)	76.34	O(3×100)

Table 5.6: The model's performances across various maximum sequence lengths. The time complexity of NAR model remain unchange as the Max sequence length change, it generate all tokens once. The time complexity of AR model is O(beam size × sequence length), each token position need to wait for the generation for all former tokens, so it will decrease as the sequence length of the output decreases.

Model Type	Time for generating 500 examples
NAR model	40.86s
AR model	510.42s

Table 5.7: Speed comparsion of AR and NAR model. The table shows the time each model need for generating 500 examples on 2080Ti GPU with batch size 8.

We noted a distinct pattern in the performance of the AR and NAR models based on varying output sequence lengths. Specifically, the performance of the AR model remains relatively stable and doesn't deviate significantly across different output lengths. In contrast, the NAR model exhibits a noteworthy increase in performance when dealing with shorter output sequences. This observation aligns with our initial assumption that the AR model inherently possesses the ability to capture long-term dependencies through

its condition mechanism. As a result, its performance remains relatively consistent despite changes in sequence length. Conversely, the NAR model's performance is more sensitive to sequence length variations, as it is more difficult for them to capture dependencies within longer sequences.

Below, In Table 5.8 and Table 5.9, we present examples of both short and long sequences generated from our model. We can see that for short sequences, both models demonstrate commendable performance. However, when faced with the complexities of longer sequences, the AR model demonstrates proficiency by producing coherent outputs. In contrast, the NAR model encounters challenges, evident in the recurrence of numerous tokens. This issue substantiates the concern previously underscored. Given the NAR model's inherent inability to exploit information from antecedent tokens, its dependence on the source input becomes more pronounced. Consequently, the NAR model tends to generate tokens that bear heightened similarity to the source input, ultimately leading to inconsistencies and resulting errors.

Source	public String toString() {return pattern();}
Reference	public override string ToString(){return Pattern();}
AR Model	public override string ToString(){return Pattern();}
NAR Model	public override string ToString(){return _pattern();}

Table 5.8: Short sentence translated by AR model and NAR model. We can see NAR model perform well can close to the AR model.

<table>
<tr><td>Source</td><td>public Object get(CharSequence key) {List<TernaryTreeNode> list = autocomplete.prefixCompletion(root, key, 0);if (list == null || list.isEmpty()) {return null;}for (TernaryTreeNode n : list) {if (charSeqEquals(n.token, key)) {return n.val;}}return null;}</td></tr>
<tr><td>Reference</td><td>public virtual object Get(string key){IList<TernaryTreeNode> list = autocomplete.PrefixCompletion(root, key, 0);if (list == null || list.Count == 0){return null;}foreach (TernaryTreeNode n in list){if (CharSeqEquals(n.token, key)){return n.val;}}return null;}</td></tr>
<tr><td>AR Model</td><td>public virtual object Get(string key){IList<TernaryTree> list = default(root, key, 0);if (list == null){return null;}for (int i = 0; i < list.Length; i++){if (char)return null;}foreach (IEnumerator n.TokenEquals(n.TokenSequence, key)){return null;}return null;}return null;}</td></tr>
<tr><td>NAR Model</td><td>public virtual object Get(string key){listist<isternTernTreeTreeNode >pleteannot =.CompleteompCompletion(rootroot, key,);.ififlist (Empty || list ||. ==)){returnreturn;} null (ListernernTreeNodeNode node in list){if (CharEquTreeEqu(alsrecogn.string,, key.)).}}return null;}</td></tr>
</table>

Table 5.9: Long sentence translated by AR model and NAR model. We can observe that NAR model has more repeated tokens and perform worse for the long sentence.

5.6 Robustness Test

An inherent challenge when employing non-autoregressive methods is determining the appropriate length for the decoder input. Unlike autoregressive methods, which can iteratively build the input by incorporating each output, non-autoregressive methods require a different approach. In our previous approach, we fixed the decoder's input length to be equal to the input sources' length as the length of the code-to-code pair is relatively close, and let the model learn to predict the EOS (end of sentence) token.

Previous research mainly relies on copy mechanisms to determine decoder input, while our approach involves utilizing the source hidden states to learn the hidden variables for decoding, incorporating guidance from target hidden states and AR models. However, when applied to tasks such as code-to-nl summarization, we face unique challenges. Notably, these challenges arise due to the substantial differences in sequence lengths between the source code and the target natural language summary, resulting in less inherent similarity between their respective hidden states. The primary challenges in this context revolve around the model's ability to accurately generate EOS tokens and address the alignment issues highlighted earlier.

In general, this project aims to develop non-autoregressive generation methods that are universally applicable across various generation tasks. While our primary focus is on code-code translation datasets, we also need to extend our evaluation to assess the compatibility of our method with code-NL summarization tasks. This additional testing is meaningful, as it addresses the challenge posed by substantial differences in source and target lengths. In contrast to translation tasks, the code-NL summarization task involves source and target sequences of very different lengths. This distinction introduces complexities in the alignment of hidden states, making the modelling of decoding hidden variables more intricate.

Model	NAR model
AR model	19.06
NAR model	12.79

Table 5.10: The performance comparison between the AR model and our top-performing NAR model on the Code-NL summarization task.

The results presented in Table 5.10 demonstrate that our model is capable of handling the code-nl task, even when dealing with differences in sequence lengths, alignment

complexities, and variations in hidden state similarity. Although our model may not yet reach the performance level of the AR model, the trade-off becomes more favourable when we consider the time complexity saved. This reduced computational burden is an essential factor in many real-world applications, making our approach practical and efficient and it's worth noting that the NAR model often refines its predicted output multiple times, which can lead to higher-quality results, but it's a trade-off between time and performance. While this additional refinement process introduces complexity, we will refrain from delving into this topic here to maintain focus on the discussion at hand.

Chapter 6

Conclusions

In this project, our focus is on investigating the benefits of fine-tuning a pre-trained LLM encoder to accelerate the inference process in Non-Autoregressive (NAR) decoding. We experiment with several techniques that aid in NAR decoding and assess the impact of integrating the LLM encoder. Then we take our best NAR model, compare its performance with the AR model, and analyze trade-offs, finally, we also evaluate the model's Robustness. Our experiment findings lead us to the following conclusions:

1. The introduction of a pre-trained encoder into the NAR model offers a more pronounced advantage compared to the AR model. We observed a Blue score improvement from 50 to 72 when incorporating a pre-trained component into the AR model. In contrast, for the NAR model, our experiments showcased a substantial enhancement from a score of 2 to 50. These findings emphasize the crucial role of pretraining in effectively capturing sequence dependencies, resulting in a robust representation that addresses a significant challenge in NAR decoding. Without the incorporation of pretraining, the NAR model would exhibit considerably poorer performance in this aspect.

2. The knowledge and contextual understanding acquired and stored in the hidden state of the pre-trained encoder yields valuable representations for NAR decoding. Consequently, conventional methods used in previous NAR studies, such as modelling hidden variables to untangle sequence dependencies and employing guidance from an AR model's hidden states, do not exhibit as significant improvements as in cases where pre-trained models are not utilized. While we do observe some enhancements in our experiments, they are not as substantial. This suggests that the necessity of modifying the hidden state or explicitly modelling hidden

variables has diminished due to the high-quality representations obtained from the pre-trained encoder.

3. The information regarding the structure and interdependency stored in the data flow, such as the origins of specific data, significantly benefits the NAR model. Even in the absence of explicit pretraining to align with this data, the model demonstrates an intrinsic ability to extract relevant information from it. Our experiments do confirm that this capability notably enhances the data flow match, thereby yielding overall improvements across various aspects.

4. The NAR model's performance improves notably as the sequence length decreases, in contrast to the AR model, which shows relatively consistent performance across different sequence lengths. Interestingly, the NAR model gains a significant speed advantage as the sequence length increases, underscoring the trade-off between performance and speed. When prioritizing performance, shorter sequences are preferable, but this leads to less noticeable speed differences between the two models. Notably, for short sequences, the NAR model performs comparably to the AR model. Furthermore, our model also exhibits commendable performance on tasks with greater complexity.

6.1 Future Work

Due to the scope of our experiment, our experiment has following limitation and give us the suggestions for future work:

1. In our NAR decoding experiment, we employed the encoder-decoder architecture. However, the decoder-only model has been gaining prominence and currently represents the state-of-the-art approach in natural language processing tasks. It is intriguing to investigate the influence of the representation learned by a fully left-to-right pre-trained decoder-only model. Experiment can be made to determine if this representation contains valuable insights and information that can benefit NAR decoding similarly to the benefits seen from the pre-trained encoder in our experiment. Additionally, the strategies to leverage this representation for faster inference times should be well designed. For instance, can we devise parameter-efficient tuning techniques that maintain an acceptable computational cost and modify the attention mask to enable a decoder-only model to perform NAR decoding effectively.

2. Our experiments demonstrated the utility of structured data, such as data flow, in providing essential dependency information that aids in NAR decoding. However, we did not pretrain with this data, which implies the model may lack awareness of the alignment between code and its corresponding data flow. Numerous studies have integrated data flow into pretraining, resulting in significant performance improvements. Moreover, apart from data flow, incorporating other low-level Intermediate Representations (IR) like Abstract Syntax Trees or LLVM can also prove beneficial. Investigating how to effectively leverage these IR representations and determining which one contributes most to enhancing our NAR model is a meaningful direction for further exploration.

Bibliography

[1] Amazon. Amazon codewhispere. https://aws.amazon.com/codewhisperer/, 2022.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs, 2019.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[6] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Akiko Eriguchi, Yoshimasa Tsuruoka, and Kyunghyun Cho. Learning to parse and translate improves neural machine translation. *arXiv preprint arXiv:1702.03525*, 2017.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[10] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[11] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. *arXiv preprint arXiv:1904.09324*, 2019.

[12] GitHub. Github copilot. https://copilot.github.com/, 2021.

[13] Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. Non-autoregressive neural machine translation. *arXiv preprint arXiv:1711.02281*, 2017.

[14] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unix-coder: Unified cross-modal pre-training for code representation, 2022.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[16] Junliang Guo, Linli Xu, and Enhong Chen. Jointly masked sequence-to-sequence model for non-autoregressive neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 376–385, 2020.

[17] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[18] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[19] Jungo Kasai, James Cross, Marjan Ghazvininejad, and Jiatao Gu. Non-autoregressive machine translation with disentangled context transformer. In *International conference on machine learning*, pages 5144–5155. PMLR, 2020.

[20] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[21] Yoon Kim and Alexander M. Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1317–1327, Austin, Texas, November 2016. Association for Computational Linguistics.

[22] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

[23] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[24] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training, 2022.

[25] Zhuohan Li, Zi Lin, Di He, Fei Tian, Tao Qin, Liwei Wang, and Tie-Yan Liu. Hint-based training for non-autoregressive machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5708–5713, Hong Kong, China, November 2019. Association for Computational Linguistics.

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[28] Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard Hovy. Flowseq: Non-autoregressive conditional sequence generation with generative flow, 2019.

[29] David J Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *International workshop on wearable and implantable body sensor networks*, 2004.

[30] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional LSTM. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 51–61, Berlin, Germany, August 2016. Association for Computational Linguistics.

[31] OpenAI. Open-ai chatgpt. https://openai.com/, 2022.

[32] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[35] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[36] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*, 2022.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[40] Shuangzhi Wu, Dongdong Zhang, Nan Yang, Mu Li, and Ming Zhou. Sequence-to-dependency neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 698–707, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[41] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[42] Kexin Yang, Wenqiang Lei, Dayiheng Liu, Weizhen Qi, and Jiancheng Lv. POS-Constrained Parallel Decoding for Non-autoregressive Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5990–6000, Online, August 2021. Association for Computational Linguistics.

Appendix A

Comparative Examples of AR and NAR Model Outputs

We can see that when sequences are short, the AR and NAR models perform similarly. But since the AR model's time increases linearly while the NAR model's time stays the same as the length increases. This means the NAR model's speed advantage is less noticeable for shorter sequences. On the other hand, for long sequence the NAR model generate output with many repeated words and incorrect tokens. This is the trade-off we talked above, An interesting idea to explore is whether breaking down longer sentences into shorter segments and using NAR decoding, and then combining them, could lead to improved performance for longer sentences while maintaining a reasonable and efficient time consumption.

```
1      public override string ToString(){return Pattern();}
3      public virtual bool contains(object o){return indexOf(o) ≠ 0;}
4      public java.nio.ByteBuffer encode(string s){return encode(java.nio.CharBuffer.wrap(java.lang.CharSequenceProxy.Wrap(s)));}
5      public override bool RequiresCommitBody(){return false;}
6      public string GetKey(){return RawParseUtils.Decode(enc, buffer, keyStart, keyEnd);}
11     public CellRangeAddress GetCellRangeAddress(int index){return (CellRangeAddress)_list[index];}
15     public virtual TreeFilter GetTreeFilter(){return treeFilter;}
17     public virtual bool canEncode(){return true;}
22     public override string ToString(){return "(" + a.ToString() + " OR " + b.ToString() + ")";}
26     public virtual FetchCommand Fetch(){return new FetchCommand(repo);}
30     public virtual void SetOutput(){output = true;}
46     public string getRawUserInfo(){return userInfo;}
47     public override object[] toArray(){lock (this._enclosing){return base.toArray();}}
49     public string getQuery(){return decode(query);}
51     public virtual ParseTreeMatch Match(IParseTree tree){return matcher.Match(tree, this);}
52     public virtual bool Contains(char[] text){return map.ContainsKey(text, 0, text.Length);}
53     public QueryRequest(string tableName){_tableName = tableName;}
55     public override bool RetryFailedLockFileCommit(){return true;}
61     public virtual E push(E @object){addElement(@object);return @object;}
63     public TreeSet(){backingMap = new java.util.TreeMap<E, object>();}
66     public NewAnalyzerTask(PerfRunData runData): base(runData){analyzerNames = new List<string>();}
67     public override bool Equals(object o){return o is EnglishStemmer;}
69     public void IncRef(){EnsureOpen();refCount.IncrementAndGet();}
71     public RefWriter(ICollection<Ref> refs){this.refs = RefComparator.Sort(refs);}
73     public virtual void EndWorker(){if (workers.DecrementAndGet() == 0){process.Release();}}
82     public SnapshotDeletionPolicy(IndexDeletionPolicy primary){this.primary = primary;}
86     public virtual CleanCommand Clean(){return new CleanCommand(repo);}
90     public static SshSessionFactory GetInstance(){return INSTANCE;}
98     public virtual void SetObjectId(AnyObjectId id){id.CopyRawTo(IdBuffer, IdOffset);}
101    public virtual void ExportDirectory(FilePath dir){exportBase.AddItem(dir);}
117    public override string ToString(){return "Provider{" + Sharpen.Util.IntToHexString(Sharpen.Util.IdentityHashCode(this)) + " " + info.name + "}";}
120    public virtual X509Certificate[] GetAcceptedIssuers(){return null;}
122    public virtual PersonIdent GetRefLogIdent(){return destination.GetRefLogIdent();}
123    public override int size(){return _size;}
125    public override string ToString(){return "I(F)";}
128    public override string GetSignerName(){return ALGORITHM_NAME;}
135    public override String ToString(){return _string.ToString();}
138    public virtual void setProgress(int progress){lock (this){setProgress(progress, false);}}
150    public virtual void SetPackedGitMMAP(bool usemmap){packedGitMMAP = usemmap;}
```

Figure A.1: Screenshot for the gold reference of the short sequence examples

```
1      public override string ToString(){return Pattern();}
3      public override bool contains(object o){return indexOf(o) ≠ -1;}
4      public java.nio.ByteBuffer encode(string s){return Encode(java.nio.CharBuffer.CharBuffer.CharBuffer.wrap(s));}
5      public override bool RequiresCommitBody(){return false;}
6      public string getKey(){return RawParseUtils.Decode(enc, buffer, bufferEnd);}
11     public CellRangeAddress GetCellRangeAddress(int index){return _list[index];}
15     public virtual TreeFilter GetTreeFilter(){return treeFilter;}
17     public override bool CanEncode(){return true;}
22     public override string ToString(){return "(" + a.ToString() + ")";}
26     public virtual FetchCommand Fetch(){return new FetchCommand(repo);}
30     public virtual void SetOutput(){output = true;}
46     public virtual string GetRawUserUserInfo(){return userInfo;}
47     public override object[] toArray(){return this._enclosing.toArrayImpl(this);}
49     public string getQuery(){return decode(query);}
51     public virtual ParseTree Match(IParseTree tree){return matcher.Match(tree, this);}
52     public virtual bool Contains(ICharSequence cs){return map.ContainsKey(cs);}
53     public QueryRequest(string tableName){_tableName = tableName;}
55     public virtual bool RetryFailedLockFileCommit(){return true;}
61     public virtual E push(E @object){addElement(@object);return @object;}
63     public TreeSet(){backingMap = new TreeMap<E, object>();}
66     public NewAnalyzerTask(PerfRunData runData): base(runData){analyzerNames = new AList<string>();}
67     public override bool Equals(object o){return o is EnglishStemmer;}
69     public void IncRef(){EnsureOpen();refCount.IncrementAndGet();}
71     public RefWriter(ICollection<Ref> refs){this.refs = RefComparator.Sort(refs);}
73     public override void EndWorker(){if (workers.DecrementAndGet();}
82     public SnapshotDeletionPolicy(IndexDeletionPolicy primary){this.primary = primary;}
86     public virtual CleanCommand Clean(){return new CleanCommand(repo);}
90     public static SshSessionFactory GetInstance(){return INSTANCE;}
98     public virtual void SetObjectId(AnyObjectId id){id.CopyRawToRaw(idBuffer, idOffset());}
101    public virtual void ExportDirectory(FilePath dir){exportBase.Add(dir);}
117    public override string ToString(){return name + " " + version;}
120    public IDATECertificate[] GetAcceptSigners(){return null;}
122    public virtual PersonIdent GetRefLogIdent(){return destination.GetRefLogIdent();}
123    public override int size(){return this._enclosing.size;}
125    public override string ToString(){return "I(F)";}
128    public virtual string GetSignerName(){return ALGitName;}
135    public override string ToString(){return _string.ToString();}
138    public virtual void setProgress(int progress){lock (this){setProgress(progress, false);}}
150    public virtual void SetPackedGitMMAP(bool usemmap){packedGitMMAP;}
```

Figure A.2: Screenshot for the AR model's outputs of the short sequence examples

```
1       public override string ToString(){return _();}}
3       public override bool contains(object o){returnreturn index(o) - -!=;}
4       public java.nio.ByteBuffer encode encode(( sjava{nn.Byte.Buffer(n..s result"; java Shar}}
5       public virtual bool RequiresCommitBody(){return false;}
6       public string GetKey(){return RawParseUtils.Decode(enc, buffer, keyStart, keyStart);}
11      public CellRangeAddress GetCellRangeAddress(int index){return _list[fieldindex];}
15      public virtual TreeFilter GetTreeFilter(){return treeFilter;}
17      public override bool canEncode(){return true;}
22      public override string ToString(){return "(" + a.ToString() + " AND " + b.ToString() + ")";}
26      public virtual FetchCommand F(){return new FetchCommand(repo);}
30      public virtual void SetOutput(){output = true;}
46      public string getRawUserInfo(){return userInfo;}
47      public override object[] toArray(){lock (._.util.objectArray}(}this
49      public string getQuery(){return decode(query);}
51      public virtual IPseTreeMatch Match(IPIPTreeTree tree){returnchercher.Match(tree,,);}
52      public virtual bool Contains(ICharSequence cs){return map.ContainsKey(cs);}
53      public QueryRequest(string tableName){_tableName = tableName;}
55      public virtual bool RetryFailedLockFileCommit(){return true;}
61      public virtual E push(E @object){addadd(@object){objectobjectobject @
63      public TreeSet(){backingMap = new java..Maputil., object>>>}
66      public NewAnalyzerTask(PerfRunData runData): base(runData){analyzerNames = new =<<<>>();}
67      public override bool Equals(object o){return o is EnglishStemmer;}
69      public void IncRef(){(){EnenssureOpen();Count.IncrementAndGet();}
71      public RefWriter(ICollectionRef> refssthis.ref = =arerator.SortSortSortsortss););
73      public override void EndWorker(){if (workers.DecmentmentAndGet() ==){}){process.Release();
82      public SnapshotDeletionPolicy(IndexDeletionPolicy primary){this.primary = primary;}
86      public virtual CleanCommand Clean(){return new CleanCommand(repo);}
90      public static SshSessionFactory GetInstance(){return INSTANCE;}
98      public virtual void SetObjectId(AnyObjectId id){id.CopyRawTo(IdBuffer, IdOffset);}
101     public virtual void ExportDirectory(FilePath dir){Base.AddAdd((,);
117     public override string ToString(){return name + " version " + version;}
120     public I XCertificateificate[]AcceptAcceptedIss(){(){return null;}
122     public virtual PersonIdent GetRefLogIdent(){return destination.GetRefLogIdent();}
123     public override int size(){return this._encl;}
125     public override string ToString(){return "I(F)";}
128     public override string GetSignerName(){return ALGORITH_NAMENAME;}
135     public override String ToString(){returnstringstring.To();}}
138     public virtual void setProgress(int progress){lock (this){set((,););}} false
150     public virtual void SetPackedGitMMAP(bool usemmap){packedGitMMAP = usemmap;}
```

Figure A.3: Screenshot for the NAR model's outputs of the short sequence examples

```
144     public virtual int Stem(char[] s, int len){for (int i = 0; i < len; i++){switch (s[i]){case 'á':s[i] = 'a';break;case 'ë':case 'é':s[i] = 'e';break;
se 'í':s[i] = 'i';break;case 'ó':case 'ő':case 'õ':case 'ö':s[i] = 'o';break;case 'ú':case 'ű':case 'ũ':case 'û':case 'ü':s[i] = 'u';break;}}len = RemoveCas
s, len);len = RemovePossessive(s, len);len = RemovePlural(s, len);return Normalize(s, len);}
145     public void AddChildBefore(EscherRecord record, int insertBeforeRecordId){for (int i = 0; i < _childRecords.Count; i++){EscherRecord rec = _childRec
ds[(i)];if (rec.RecordId == insertBeforeRecordId){_childRecords.Insert(i++, record);}}}
146     public ListAlbumsRequest(): base("CloudPhoto", "2017-07-11", "ListAlbums", "cloudphoto", "openAPI"){Protocol = ProtocolType.HTTPS;}
147     public SaveTaskForUpdatingRegistrantInfoByIdentityCredentialRequest(): base("Domain-intl", "2017-12-18", "SaveTaskForUpdatingRegistrantInfoByIdentit
redential", "domain", "openAPI"){Method = MethodType.POST;}
148     public override ValueEval Evaluate(int srcRowIndex, int srcColumnIndex, ValueEval arg0){int result;if (arg0 is TwoDEval){result = ((TwoDEval)arg0).H
ght;}else if (arg0 is RefEval){result = 1;}else{ return ErrorEval.VALUE_INVALID;}return new NumberEval(result);}
149     public virtual DescribeReservedInstancesResponse DescribeReservedInstances(){return DescribeReservedInstances(new DescribeReservedInstancesRequest()
}
150     public virtual void SetPackedGitMMAP(bool usemmap){packedGitMMAP = usemmap;}
151     public POIFSDocumentPath(){this.components = new string[0];}
152     public override string ToString(){return Key + "/" + Value;}
153     public override void Decode(byte[] blocks, int blocksOffset, int[] values, int valuesOffset, int iterations){for (int i = 0; i < iterations; ++i){in
byte0 = blocks[blocksOffset++] & 0xFF;int byte1 = blocks[blocksOffset++] & 0xFF;int byte2 = blocks[blocksOffset++] & 0xFF;values[valuesOffset++] = (byte0 <<
2) | (byte1 << 4) | ((int)((uint)byte2 >> 4));int byte3 = blocks[blocksOffset++] & 0xFF;int byte4 = blocks[blocksOffset++] & 0xFF;values[valuesOffset++] = (
yte2 & 15) << 16) | (byte3 << 8) | byte4;}}
154     public void Serialize(ILittleEndianOutput out1){out1.WriteShort(_extBookIndex);out1.WriteShort(_firstSheetIndex);out1.WriteShort(_lastSheetIndex);}
155     public PatternParser(IPatternConsumer consumer): this(){this.consumer = consumer;}
156     public string[] GetValues(string name){var result = new List<string>();foreach (IIndexableField field in fields){if (field.Name.Equals(name, StringC
parison.Ordinal) && field.GetStringValue() != null){result.Add(field.GetStringValue());}}if (result.Count == 0){return NO_STRINGS;}return result.ToArray();}
157     public virtual ListIdentityPoolUsageResponse ListIdentityPoolUsage(ListIdentityPoolUsageRequest request){var options = new InvokeOptions();options.R
uestMarshaller = ListIdentityPoolUsageRequestMarshaller.Instance;options.ResponseUnmarshaller = ListIdentityPoolUsageResponseUnmarshaller.Instance;return In
ke<ListIdentityPoolUsageResponse>(request, options);}
158     public ValueEval Evaluate(ValueEval[] args, int srcCellRow, int srcCellCol){if (args.Length < 3 || args.Length > 5){return ErrorEval.VALUE_INVALID;}
y{BaseRef baseRef = EvaluateBaseRef(args[0]);int rowOffset = EvaluateIntArg(args[1], srcCellRow, srcCellCol);int columnOffset = EvaluateIntArg(args[2], srcC
lRow, srcCellCol);int height = baseRef.Height;int width = baseRef.Width;switch (args.Length){case 5:width = EvaluateIntArg(args[4], srcCellRow, srcCellCol);
eak;case 4:height = EvaluateIntArg(args[3], srcCellRow, srcCellCol);break;}if (height == 0 || width == 0){return ErrorEval.REF_INVALID;}LinearOffsetRange ro
ffsetRange = new LinearOffsetRange(rowOffset, height);LinearOffsetRange colOffsetRange = new LinearOffsetRange(columnOffset, width);return CreateOffset(base
f, rowOffsetRange, colOffsetRange);}catch (EvaluationException e){return e.GetErrorEval();}}
159     public virtual int[] GetCountsByTime(){return countsByTime;}
```

Figure A.4: Screenshot for the gold reference of the short sequence examples

```
144     public virtual int Stem(char[] s, int len){for (int i = 0; i < len; i++){switch (s[i])){case 'e':case 'e':case 'e':case 'e':case 'e':case 'e':case '
:case 'e':case 'e':case 'e':case 'e':case 'ö':case 'o':case 'ö':case 'ö':case 'o':case 'o':case 'o':case 'o':case 'o':case 'ö':case 'o':case 'ö':case 'o':ca
 'o':case 'o':case 'o':case 'o':case 'o
145     public void AddChildBefore(EscherRecord record, int insertBeforeRecord){int idx = 0;for (int i = 0; i < this; i++){if (rec.RecordId == (rec.RecordId
break;}_childRecords.Add(idx, record);_childRecords.Add(idx, record);}_childRecords.Add(idx, record);}
146     public ListAlbumsRequest(): base("CloudPhoto", "2017-07-11", "ListAlbums", "cloudphoto", "openAPI"){Protocol = ProtocolType.HTTPS;}
147     public SaveTaskForUpdatingRegistryRequest(): base("Domain-intl", "SaveTaskForUpdatingRegistrantInfoByIdentityCredential", "openAPI"){Method = Method
pe.POST;}
148     public ValueEval Evaluate(int srcRowIndex, int srcColumnIndex, ValueEval arg0){int result;if (arg0 is RefEval){result = ((TwoDEval)arg0).GetHeight()
else{result = 1;return ErrorEval.VALUE_INVALID;}return new NumberEval(result);}
149     public virtual DescribeReservedInstancesResponse DescribeReservedInstances(){return DescribeReservedInstances(new DescribeReservedInstancesRequest()
}
150     public virtual void SetPackedGitMMAP(bool usemmap){packedGitMMAP;}
151     public POIFSDocumentPath(){this.components = new string[0];}
152     public override string ToString(){return key + "/" + value;}
153     public override void Decode(byte[] blocks, int blocksOffset, int[] values, int valuesOffset, int iterations){for (int i = 0; i < iterations; ++i){in
byte0 = blocks[blocksOffset++] & 0xFF;int byte1 = blocks[blocksOffset++] & 0xFF;values[blocksOffset++] = blocks[blocksOffset++] & 0xFF;valuesOffset++] = bloc
s[valuesOffset++] & 0xFF;values[valuesOffset++] = blocks[valuesOffset++] & 0xFF;values[valuesOffset++] = ((byte3 & 0xFF;}}
154     public override void Serialize(ILittleEndianOutput out1){out1.WriteShort(_extBookIndex);out1.WriteShort(_firstSheetIndex);out1.WriteShort(_firstShee
ndex);}
155     public PatternParser(PatternConsumer consumer){this.consumer = consumer;}
156     public string[] GetValues(string name){IList<string> result = new List<string>();foreach (IIndexableField field in fields){if (field.Equals(name.Ord
al)){result.Add(field.Name, StringComparison.Ordinal)){result.Add(field.Name);}}return NO_STRING;}
157     public virtual ListIdentityPoolUsageResponse ListIdentityPoolUsage(ListIdentityPoolUsageRequest request){var options = new InvokeOptions();options.R
uestMarshaller = ListIdentityPoolUsageRequestMarshaller.Instance;options.ResponseUnmarshaller = ListIdentityPoolUsageResponseUnmarshaller.Instance;return In
ke<ListIdentityPoolUsageResponse>(request, options);}
158     public ValueEval Evaluate(ValueEval[] args, int srcCellRow, int srcCellCol){if (args.Length < 1){return ErrorEval.VALUE_INVALID;}try{return ErrorEva
VALUE_INVALID;}try{baseRef = EvaluateBase.VALUE_INVALID;int rowOffset = EvaluateRef(args[0], srcCellRow, srcCellRow, srcCellCol);int rowOffset = EvaluateInt
(args, srcCellCol);int rowOffset = EvaluateInt32(args[0], srcCellRow, srcCellCol);if (args.Length > 0){width = EvaluateInt32(args, srcCellCol
159     public virtual int[] GetCountsByTime(){return countsByTime;}
```

Figure A.5: Screenshot for the AR model's outputs of the short sequence examples

```
144     public virtual int Stem(char[] s, int len){for (int i = 0; i < len; i++){switch (s[i]){caseså':s[i] = 'a';break;case 'é':case 'é':s[i] = 'e';break;
se 'ö':s[i] = 'i';break;case 'ö':casecase�ú':case''ö':case 'ö':s[i] = 'o';break;case 'ú':case''ü':case '�ú':case 'ú':case 'ö':s[i] = 'u';break; Removes( =,)
;break new Removeoss(sess,,sens,s Remove((s = lenizes, lensize, Stizes( len,}
145     public void AddChildBefore(EscherRecord record, int insertBeforeRecordId){int idx = 0;for (IEnumer.Record in iterator.GetEnumer..Move();){iteratorRe
rd)RecordBeforeRecord.Recordrecbreak){Record;}RecordRecordbreakRecords.childAdd.idxlist);;}
146     public ListAlbumsRequest(): base("CloudPhoto", "2017-07-11", "ListAlbums", "cloudphoto", "openAPI"){Protocol = ProtocolType.HTTPS;}
147     public SaveTaskForUpdatingRegistrantInfoByIdentityCredentialRequest(): base("Domain-intl", "2017-12-18", "SaveTaskForFordatingForUpistRegInfoentityI
oredential", "domain", "openAPI"){Method = MethodType.POST;}
148     public override ValueEval Evaluate(int srcRowIndex, int srcColumnIndex, ValueEval arg0){int result;if (arg0 is AreaEval){result = ((AreaEval)arg0).H
ght;}else if (arg0 is RefEval){result = 1;}else{ return ErrorEval.VALUE_INVALID;}return new NumberEval(result);}
149     public virtual DescribeReservedInstancesResponse DescribeReservedInstances(){return DescribeReservedInstances(new DescribeReservedInstancesRequest()
}
150     public virtual voidPPackedGitMMAP( bool usemmap){packedGitMMAP = usemmap;}
151     public POIFSDocumentPath(){this.components = new string[ 0];}
152     public override string ToString(){return key + "/" + value;}
153     public override void Decode(byte[] blocks, int blocksOffset, int[] values, int valuesOffset, int iterations){for (int i = 0; i < iterations; ++i){in
byte0 = blocks[blocksOffset++] & 0xFF;int byte1 = blocks[blocksOffset++] & 0xFF;int byte2 = blocks[blocksOffset++] & 0xFF;values[valuesOffset++] = (byte0 <<
2) | (byte1 << 4) | ((int)((uintongbytebyte >> >>));int byte3 = blocks[[Offset++++ & 0xx;int byte44 blocks[blocksOffset++]] 0xFF;values[valuesOffset++++ = (
ytebyte2 15) << 16) | (byte3 << 8) | byte4;}}
154     public void Serialize(ILittleEndianOutput out1){out1.WriteShort(_extBookIndex);out1.WriteShort(_(_SheetetIndex);out1.ShortShortSheSheetIndex);}
155     public PatternParser(PatternConsumerconsumer): this(){this..consumer.
156     public string[] GetValues(string name){var result<> result = new List<<string>>foreachIIndexableField field in field in fields.field.Equ.(name,Compa
 StringComparison.Ordinalison && field..ValueField.Value..Value(inal))}}}}if.){ ==if (._.returnString;}
157     public virtual ListIdentityPoolUsageResponse ListIdentityPoolUsage(ListIdentityPoolUsageRequest request){var options = new InvokeOptions();options.R
uestMarshaller = ListIdentityPoolUsageRequestMarshaller.Instance;options.arsharsharsharshResponseUnarsharshUnarshUnUnarsharsharshUnallUnarsharsharshUnUnarsh
InstancearsharshUnarshUnarsharsharsharsharsharshUnarsharshokeUnokearshUnarshUnarsharsharshUnallarshUnarshUnUnUnUnUnarsharsharsharshResponseUnResponsearshU
rsharsharsharshUnUnarsharshokearsharshUnarshUnUnarsharshResponseUnarsharsharsharshUnarshUnarshUnarsharshUnallarshUnarsharsharsharsharshUnarshResponseUna
hUnUnarshUnUnarsharsharsharsharshResponsearsharsh.arshUnUnarsharshUnarshUnarshUnallUnarsharsharsharsharsharshUnarshUnarsharshUnarsharshUnokearshUnUnarsharsh
shUnUnarshUnUn
158     public ValueEval Evaluate(ValueEval[] args, int srcCellRow, int srcCellCol){if (args.Length < 1 || args.Length > 5){return ErrorEval.VALUE_INVALID;}
y{RowRefRefRef = EvalBaseRef(args[0]);int src = = ([[args1 == Missing isArgEval)0 0 :uateIntInt(Arg[args], srcRowRowCellCellRow);CellCellOffset); (firstOffs
 (args[[ 2ArgArgval)] == 0argsIntInt(args[22Arg srcCellCol, srcRange);int = heightintRef =intWidth;int;} = baseRef = 2Widthintwidth = (switchswitch: (case 5
f ( (args.[argsValue))throwreturn ArgumentArgArg{[ = =uateInt((args);iColumnRow srcCellRow, srcRowColCol;}:: if ( ((.Args[[Arg[ArgArgargs{[ = =(:(args,,3Cell
w srcRow);return srcRange);Cell
159     public virtual int[] GetCountsByTime(){return countsByTime;}
```

Figure A.6: Screenshot for the AR model's outputs of the short sequence examples