Learning A Transferable Code Model With Global Information Based On A New Paradigm

Anonymous ACL submission

Abstract

Companies like Microsoft Research and Google DeepMind have highlighted limitations in GPTs' next-word prediction approach, such as poor planning, memory, and reasoning. GPTs generate text locally without global task understanding, struggling with complex logic and unseen code generation, as confirmed by our code comprehension studies. We propose a new heterogeneous image paradigm for code encoding, inspired by diffusion techniques in image and protein structure generation. This approach encodes code globally, combining image and protein-like structures, avoiding autoregressive constraints. Using a CLIP-inspired text-to-code encoder, the model maps text to code for various tasks. Trained on 456,360 text-code pairs with self-supervised learning, the model achieves zero-error predictions, bridging text and code encoding spaces. This work paves the way for diffusion-based code generation, overcoming GPTs' limitations.

1 Introduction

Studies highlight limitations in GPTs' autoregressive next-word prediction paradigm. Microsoft Research notes GPT-4 lacks planning, memory, backtracking, and reasoning skills, often requiring step-by-step guidance to provide correct answers despite having sufficient knowledge (Bubeck et al., 2023). Google DeepMind adds that GPTs struggle in mathematics due to the high cost of converting human proofs into a machine-verifiable format (Trinh et al., 2024). Rearranging premise order can reduce performance by over 30% (Chen et al., 2024), and GPT-4 fails to solve IMO geometry problems, producing errors and showing poor understanding (Bubeck et al., 2023). In summary, GPTs rely on a local, greedy process, lacking global task understanding.

Our empirical studies in code understanding confirm GPT-4's limitations under the autoregressive paradigm. It struggles with complex logic and multi-step operations, often generating incomplete or buggy code. GPT-4 is overly sensitive to prompt phrasing—even semantically equivalent inputs with different syntax can lead to inconsistent and error-prone outputs. It also performs poorly in matrix operations, frequently introducing new bugs while attempting fixes. While step-by-step guidance can elicit correct answers, indicating sufficient training knowledge, the autoregressive nature hinders its ability to generate complete, correct solutions for multi-step or matrix operations in one go. These findings align with Microsoft and Google's observations, highlighting GPT-4's reliance on prompts, lack of global code understanding, and inability to handle complex tasks effectively.

Diffusion technology has advanced significantly in image generation (e.g., DALL·E 2, Sora) and life molecule modeling (e.g., AlphaFold 3) (Ramesh et al., 2022; Peebles and Xie, 2023; Josh Abramson, 2024). Unlike GPTs, diffusion models learn global information and generate outputs (e.g., images or molecules) all at once, avoiding step-by-step limitations. Inspired by non-equilibrium thermodynamics, diffusion models, such as CLIP+diffusion (DALL·E 2), create novel, high-quality images from text inputs (e.g., "panda mad scientist mixing sparkling chemicals") (Ramesh et al., 2022; Peebles and Xie, 2023; Ho et al., 2020). CLIP, a key component, bridges text and images by embedding them into a shared semantic space, aligning similar concepts (Radford et al., 2021). This enables the diffusion model to generate images from text embeddings, demonstrating a deep understanding of the input.

We propose applying the CLIP+Diffusion technique to code generation to overcome the limitations of the autoregressive paradigm. Code shares properties with both natural language and images. Like natural language, code uses tokens and follows grammatical rules, but unlike natural lan-

guage, code has minimal ambiguity and avoids out-of-vocabulary (OOV) issues through namespace partitioning. This suggests that code generation may not need the next-word prediction approach used in autoregressive models. Additionally, code resembles images in its heterogeneity. Just as images are composed of components like colors (e.g., yellow, blue) with variations (e.g., light yellow, dark yellow), code consists of components like classes, methods, and variables, each containing distinct entities (e.g., method 1, method 2). This structural similarity makes diffusion models, which excel at capturing global information, a promising alternative for code generation.

We propose a transferable pre-training CLIP model for code comprehension, serving as the foundation for high-quality code generation using diffusion technology and other tasks. To encode code, we break away from GPTs' autoregressive, natural language-like approach and introduce a single-channel, one-dimensional, heterogeneous image paradigm, inspired by image encoding. This aligns with the sequential nature of human programming. Built-in or common code components (e.g., classes, methods, operators) are encoded as fixed numerical IDs, with similar entities (e.g., Python's 'print()' and 'len()') assigned similar IDs, mimicking pixel similarity in images. User-defined components have unique IDs within their namespace, with similar entities also encoded similarly. This approach avoids out-of-vocabulary (OOV) issues and eliminates the need for token embeddings, ensuring the paradigm mimics images, not natural language. For pre-training, we design a code encoder using one-dimensional convolution and local pooling, tailored to this encoding scheme. Unlike GPTs, which predict the next token locally, our encoder iteratively learns global code information, enabling a deeper understanding of code structure and semantics.

We validated the proposed method with limited data, observing performance improvements as data increased. Through ablation experiments, we identified optimal model designs. While large-scale training and direct application were beyond our resources, this work aims to offer innovative ideas for advancing programming language understanding.

2 Motivation

GPTs' autoregressive paradigm has limitations such as lack of planning, working memory, and reasoning abilities, hindering their application in many areas, including mathematics (Bubeck et al., 2023; Trinh et al., 2024). Researchers attribute this to their reliance on local, greedy next-word generation. Our case studies in code understanding with GPT-4 further confirm these limitations.

2.1 Cases Studies

Lack of ability to handle complex logic. We asked GPT-4 about a code problem involving complex logic and multi-step operations, summarizing it as: "We want to train an embedding network with input shape (359, 4), where each of the 359 arrays contains 4 lists, and output an array of the same shape with embedded lists of length n. The label has shape (359,)." Although humans can grasp the problem, GPT-4 provided incomplete and logically flawed code. For instance, GPT-4 misunderstood the desired output shape and incorrectly concatenated the lists. Additionally, the training process was incomplete, lacking a loop for epochs, and variables were undefined (indicated by red wavy lines).

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Flatten, Dense

input_shape = (4,)
embedding_dim = n
input_layer = Input(shape=input_shape)
embedded_lists = []
for i in range(4):
    embedded_list = Embedding(input_dim=input_shape[i],
                              output_dim=embedding_dim)(input_layer[i])
    embedded_list = Flatten()(embedded_list)
    embedded_lists.append(embedded_list)

concatenated_embedding = Concatenate()(embedded_lists)
output_layer = Dense(4)(concatenated_embedding)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(optimizer='adam', loss='mse')
model.fit(x=training_data, y=labels, epochs=num_epochs,
          batch_size=batch_size)
```

Figure 1: An example of GPT-4 handles complex logic.

Lack of working memory. If we ask the same question again after a few dialogue rounds because we forgot the first answer, GPT-4 will often give a completely different answer. This results in almost all the dialogue rounds being invalid. For example, we repeated the GPT-4 question "How do I use a terminal command to issue a commit that changes more than n files in a repository?" twice, with several rounds of dialogue in between. GPT 4 gave two different incorrect answers, as shown in Fig. 2 (a) and (b) respectively.

Overly dependent on prompt format. GPT-4 will give different answers to questions with different formats but the same meaning. For example,

```
git log --stat --numstat --oneline | awk 'NF==1 {commit=$1} NF>1
{changed_files[commit]+=$1+$2} END {for (commit in changed_files)
if (changed_files[commit] > 5) print commit}'
```

(a) First answer

```
git log --name-only --oneline | awk '/^commit/ {if (n>0) print c
" " n; n=0; c=$1} /^ [MADRCUT]/ {n++}' | awk '$2 > n' n=5
```

(b) Second answer

Figure 2: An example of GPT-4 lacking memory.

if we ask "How do I use a terminal command to find the commits that change more than n files in a repository?" and "How do I use a terminal command to find the commits with multiple changed files greater than n in a repository? GPT-4 gives very different answers as shown in Fig. 3.

```
git log --stat --numstat --oneline | awk 'NF==1 {commit=$1} NF>1
{changed_files[commit]+=$1+$2} END {for (commit in changed_files)
if (changed_files[commit] > 5) print commit}'
```

(a) First answer

```
git log --format="%H" --numstat | awk '/^$/ {commit=0} {if(
commit==1) {print $1} commit=1}' | while read commit; do git diff
--name-only $commit^ $commit | wc -l | tr -d '[:space:]'; done |
awk -v n=5 '$1 > n {print $2}'
```

(b) Second answer

Figure 3: An example of GPT-4 over-relying on the prompt format.

Not good at matrix operations. GPT-4 often fails to handle matrix operations correctly or introduces new bugs when handling matrix operations. For example, as shown in Fig. 4(a), when we asked GPT-4 why the figure (a) code reported an error: "ValueError: could not broadcast input array from shape (359,1149) into shape (359,)". GPT-4 will allow us to modify according to Fig. 4(b). But it introduces new bugs into the code: *ValueError: only one element tensors can be converted to Python scalars.* GPT-4 ignores global correctness and instead gets stuck in solving local problems.

```
def forward(self, x):
    data = np.zeros((x.size(0), 4))
    data[:, 0] = x[:, :1149]
    data[:, 1] = x[:, 1149:4149]
    data[:, 2] = x[:, 4149:7149]
    data[:, 3] = x[:, 4149:7167]
```

(a) Problem code

```
def forward(self, x):
    data = np.zeros((x.size(0), 4))
    data[:, 0] = x[:, :1149]
    data[:, 1] = x[:, 1149:4149]
    data[:, 2] = x[:, 4149:7149]
    data[:, 3] = x[:, 7149:7167]
```

(b) Fixed code

Figure 4: An example of GPT-4 is not good at matrix operations.

Lack of creativity. GPT-4 is not creative. For example, if we ask GPT-4 to invent an algorithm for predicting earthquakes, GPT-4 cannot provide an answer, even if it is to give some ideas for this scientific question.

2.2 Discussion

CLIP+Diffusion in Image Understanding/Generation can mitigate limitations similar to those of GPTs in code tasks. As illustrated in Fig. 5(a) and (b), CLIP+Diffusion globally comprehends complex text input and accurately generates images with intricate structures. The lines and colors in (a) are intricate and smooth, while the fur and textures in (b) are clearly rendered. This method remains creative and accurate even when the input text's form varies but the meaning stays the same. As shown in Fig. 5(c) and (d), CLIP+Diffusion creates unprecedented images like *panda mad scientist* and *a cat dressed as French Emperor Napoleon holding cheese*, demonstrating its innovative capability. Thus, we believe applying this technology to code understanding/generation could significantly enhance performance in this area.

(a) (b) (c) (d)

Figure 5: Examples of CLIP+Diffusion can avoid some limitations similar to GPTs.

Why GPTs Have Limitations in Code Understanding/Generation Compared to CLIP+Diffusion We observed that the CLIP model globally remembers all image information during training, unlike GPTs which predict the next token based solely on previous code. For instance, CLIP learns the global distribution of fur color, texture, and facial feature structure in multiple Shiba Inu images. The image representation corresponding to human-given textual descriptions already contains comprehensive global image information (Ramesh et al., 2022). Diffusion then refines this representation into a high-quality image. However, GPTs, adhering to the next-token prediction paradigm, cannot globally retain fully

functional code information. As illustrated in Fig. 1, GPT-4 struggles to generate structurally complete code even if it has learned complex code snippets. This paradigm also leads to other limitations. For instance, GPT-4 may provide vastly different answers to the same question asked multiple times due to changes in preceding text. Similarly, questions with the same meaning but different wording may elicit varied responses from GPT-4. Furthermore, GPTs' lack of proficiency in handling matrix operations and limited creativity stem from their inability to truly understand code. Comprehensive code understanding requires global learning of code information during training, not just local context. We believe that GPTs' autoregressive next-word prediction paradigm is the root cause of these limitations.

Why GPTs Use the Autoregressive Paradigm for Code GPTs utilize the autoregressive paradigm for understanding and generating code primarily because it is well-suited for natural language processing. Both code and natural language serve as communication tools, with code facilitating human-computer interaction and natural language enabling human-to-human communication. Both consist of token sequences governed by syntax rules. However, natural language exhibits greater flexibility than code, with word meanings varying across contexts. Thus, understanding and generating natural language necessitates rigorous contextual analysis, driving GPTs to adopt the autoregressive paradigm. This approach ensures semantic correctness by generating tokens sequentially, relying on previous text at each step. Transformer decoders, employed by GPTs, adhere to this paradigm during generation. Although Transformer encoders can globally learn text information, the decoder focuses on autoregressive token prediction. We summarize this analysis in Table 1, columns 1 and 3.

Challenges and Solutions in Proposing a New Paradigm for Code To overcome the limitations of GPTs, we propose a structured coding paradigm that mimics images, aiming to leverage CLIP+Diffusion technology. However, code exhibits both image-like and natural language-like features, posing a challenge. We observe similarities between code and images in terms of component structure (Table 1, columns 2 and 4). Codes consist of entities like classes, methods, and variables, akin to image components like colors and shapes. This heterogeneity—where an object is composed of different types of components—is shared by both. However, code tokens face an out-of-vocabulary (OOV) problem due to their infinite variety, unlike the finite vocabulary of natural language. This hinders the application of CLIP+Diffusion. If natural language embedding techniques are used, they fail to address this issue. Nevertheless, code entities' namespaces are often smaller and more scoped than natural language words. Code namespaces are typically related to packages, folders, files, or code snippets, whereas natural language words often span the entire vocabulary. By strictly partitioning namespaces and using repeated numbers to encode cross-namespace tokens, we may resolve the OOV problem. This approach leverages the scoped nature of code namespaces to manage token representation effectively.

	Linguistic	Heterogeneous	Ambiguous	OOV
Image	×	✓	×	×
Code	✓	✓	×	Solvable
Text	✓	×	✓	✓

Linguistic: Is it language?
Heterogeneous: Is it heterogeneous?
Ambiguous: Is it possible to have ambiguity?
OOV: Is it has OOV issues?

Table 1: The similarity of image, text, and code

3 Approach

3.1 Heterogeneous image paradigm

Proposed Heterogeneous Image Paradigm for Code. Given code's standardized structure, we abandon the autoregressive *next-word prediction* paradigm and propose a single-channel, one-dimensional *heterogeneous image* paradigm for code understanding and generation. Both code and images share heterogeneity, composed of various components. We treat code components (classes, methods, variables, operators, numbers, symbols) analogously to image components (colors). Entities within the same code component are represented by similar numerical values (IDs), mimicking pixel values within the same image component. To address the OOV problem due to numerous tokens, we take several steps: 1)Optional cleanup of code by replacing messy data with placeholders to ensure accuracy in text-to-code matching. 2)Classification of tokens into built-in (code library) and user-defined types. 3)Further division of user-defined tokens into namespaces. 4)Representation of built-in tokens by fixed numeric values (IDs). 5)Representation of user-defined tokens by temporary, reusable values within their namespaces. As illustrated in Fig. 6(a), a function is depicted as a 2D heterogeneous image, where keywords, variables,

Components	ID Range	Components	ID Range
Keyword	1-35	Built-in Attribute	6930-7960
Built-in Class	36-54	Variable	7961-9999
Class	55-1584	Built-in AttrCall	10000-11270
Built-in Method	1585-2698	Attribute Call	11271-11509
Method	2699-4454	Operator	11510-11554
Built-in MethCall	4455-6128	Number	11555-13811
Method Call	6129-6929		

AttrCall: Attribute call, MethCall: Method call.

Table 2: Components and their ID ranges

and operators/symbols are represented by similar pixel values in red, green, and yellow, respectively.

```
def maxNum (num1, num2):
    if num1 > num2:
        return num1
    else:
        return num2
```

(a) 2-dimensional heterogeneous image

```
def maxNum (num1, num2): \n Space if num1 > num2: ...
```

(b) 1-dimensional heterogeneous image

Figure 6: Examples of heterogeneous image.

Abstracting Entity Call Tokens. In addition to the process outlined in steps 1)-5), we apply the same methodology to tokens representing entity calls (e.g., method calls or class attribute accesses). To minimize the number of these tokens and mitigate OOV issues, we abstract them. Specifically, tokens containing the same entity call (e.g., *a_param.strip* and *address.strip*) share the same numeric value. We maintain a list for each numeric value to facilitate manual lookup of the specific token it represents. Using the encoding method detailed above, we constructed a Python vocabulary based on the 456,360 code snippets in the CodeSearchNet corpus's training set[1]. This corpus comprises code snippets and their associated documentation comments from various programming languages sourced from open-source GitHub repositories. GitHub initiated and maintains this corpus to foster the development of code search and understanding technologies. The number of tokens and their corresponding IDs vary across programming languages. For instance, Tab. 2 showcases the specific components and their numerical coding ranges in Python.

Color Images vs. Heterogeneous Image Representation Color images are composed of three primary colors: red, blue, and green, forming three channels. In contrast, the heterogeneous image introduced in this paper features a single channel, where different components hold equal significance, unlike the hierarchical composition in color images.

[1]https://opendatalab.org.cn/CodeSearchNet

Sequential Nature of Code Human programming involves writing code sequentially. Hence, the dependency between tokens is stronger horizontally (left-to-right) than vertically. Consequently, the two-dimensional heterogeneous image is flattened into a one-dimensional form, retaining line breaks and spaces as symbols, akin to Fig. 6(b). This structure resembles a biological molecule chain, such as proteins, DNA, or RNA, composed of basic elements and exhibiting heterogeneity.

Inspiration from Diffusion Technology DeepMind's application of diffusion technology to predict living molecule structures inspires us to explore its potential in code understanding and generation. A foundational task is designing a robust pre-trained model for text-to-code conversion.

Code as a Hybrid of Images and Language Code shares similarities with both images and natural language. The proposed paradigm mimics the numerical encoding of image pixels for code tokens but retains a sequential structure akin to natural language.

3.2 Contrastive Language-Code Pre-training

3.2.1 Architecture

We propose the Contrastive Language Code Pre-training (CLCP) model, inspired by the Contrastive Language Image Pre-training (CLIP) model (Radford et al., 2021). CLCP jointly trains a code encoder and a text encoder to predict correct (code, text) pairs in a batch. As shown in Fig. 7, given a batch of N (code, text) pairs, CLCP predicts the correct pairings among $N \times N$ possibilities, learning a multimodal embedding space. We use a pre-trained text transformer as the text encoder (Vaswani et al., 2017; Hu et al., 2020), leveraging its strengths in natural language understanding (Vaswani et al., 2017; Devlin et al., 2019; Brown et al., 2020; Ouyang et al., 2022). For the code encoder, we design a new architecture tailored to our proposed code encoding paradigm. The model optimizes a symmetric cross-entropy loss, maximizing the cosine similarity $C_i \cdot T_i$ $(1 \leq i \leq N)$ of correct pairs while minimizing $C_i \cdot T_j$ $(1 \leq i, j \leq N, i \neq j)$ for incorrect pairings (Radford et al., 2021). This approach enables future downstream tasks, such as using the text encoder to generate code via diffusion models or translating code across programming languages using code encoders.

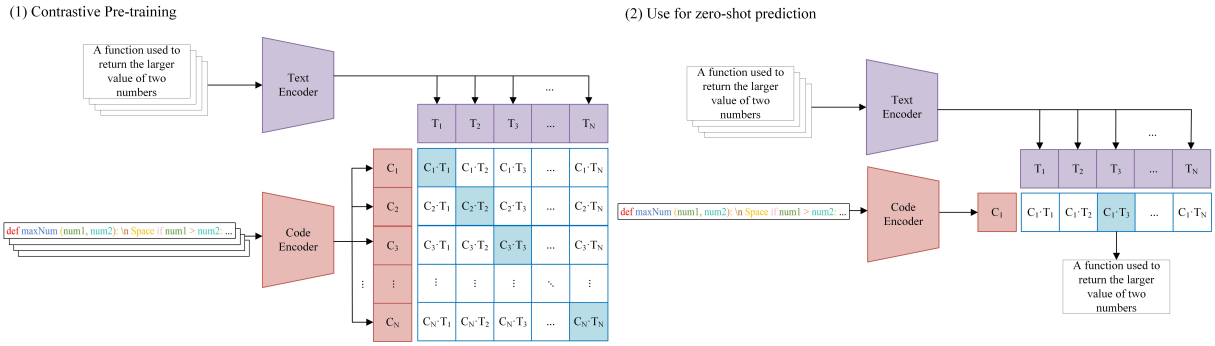

Figure 7: Architecture. CLCP jointly trains a code encoder and a text encoder to predict the correct(code, text) pairs. In the testing phase, for the target code, the learned model is treated as a zero-shot classifier to select the description that best matches it.

3.2.2 Code encoder based on one-dimensional convolution

For the code encoder, we adapt two-dimensional convolution for images (LeCun et al., 1998) to a one-dimensional convolutional and pooling neural network, suitable for one-dimensional heterogeneous images. As shown in Fig. 8, the source code is first encoded into one-dimensional heterogeneous images using the new paradigm. Next, one-dimensional convolution is applied with a kernel size k and step size s to learn the semantic and structural information of the code. The convolution kernel slides over the one-dimensional image, multiplying IDs by corresponding weights and summing them to produce a new feature map. Each value in the feature map is activated using the ReLU function. Then, one-dimensional (1D) maximum or average pooling is applied with window size k' and step size s'. Pooling divides the input vector into sub-vectors and computes the maximum or average value of each sub-vector, aggregating information from sub-code fragments. The code encoder consists of M blocks, each containing a convolution layer and a pooling layer, designed to learn and aggregate code semantics and structure effectively. The artifacts are archived at: https://doi.org/10.5281/zenodo.13148594. This work is licensed under the Apache License, Version 2.0.

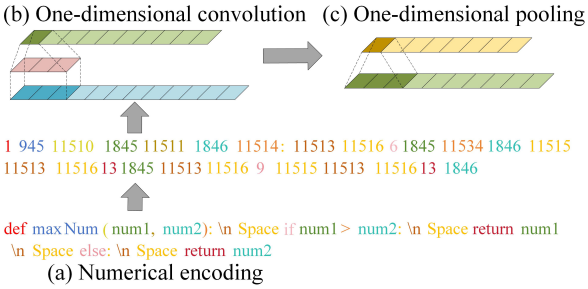

Figure 8: The specific implementation of a block of the code encoder.

4 Experiments

4.1 Research Questions

We evaluate the model's effectiveness on the following question using 4×NVIDIA Tesla V100 with 32GB RAM:

RQ1. How effective is CLCP at zero-shot transfer? Given limited training data (456,360 pairs) compared to large datasets (e.g., 400 million pairs for CLIP), achieving high accuracy in zero-shot tasks is challenging. Previous works using natural language supervision for representation learning, similar to CLIP (Li et al., 2017; Desai and Johnson, 2021; Sariyildiz et al., 2020; Zhang et al., 2022a), showed limited performance due to data constraints (e.g., 11.5% accuracy on ImageNet (Li et al., 2017)). Collecting high-quality data remains difficult. Thus, we adjust training and testing set sizes to assess the model's impact on zero-shot tasks and its ability to learn global code information.

RQ2. Ablation Experiment. We conducted ablation experiments to explore specific training strategies for learning code-heterogeneous images. Specifically, we investigated the impact of pooling operations, which reduce feature map dimensionality by aggregating information, potentially leading to information loss. To assess this, we removed the pooling layer from the CLCP model. Furthermore, we employed He initialization (He et al., 2015) to mitigate gradient explosion in backpropagation and omitted Batch Normalization (BN) to preserve input image distribution characteristics. To verify these choices, we conducted experiments by removing He initialization and adding BN, respectively.

4.2 Dataset

To reduce preprocessing efforts, this paper evaluates the performance using Python as an example. We utilize the Python dataset from the CodeSearchNet corpus[2], which contains 456,360 (code, text) pairs in the training set and 22,176 pairs in the test set. The CodeSearchNet corpus comprises several public code libraries from GitHub, spanning various programming languages such as Java, Python, and JavaScript. These libraries include numerous code snippets from diverse open-source software repositories, with corresponding document comments aiding in understanding code functionality. Initiated and maintained by GitHub, this corpus aims to promote code understanding technologies. For zero-shot task evaluation, we selected 13,760 unique samples from the test set, verified through text description analysis and manual checks, ensuring no overlap with the training set categories.

4.3 Experimental Setup

Since the amount of data is limited, we imitate the works (Li et al., 2017; Desai and Johnson, 2021; Sariyildiz et al., 2020; Zhang et al., 2022b) referred to in the clip paper to do some preliminary exploration to verify that the proposed approach is effective (e.g. the accuracy in (Li et al., 2017) is only 11.5%), instead of pursuing the effect of training under 400 million data as in the clip model.

4.3.1 Settings for RQ1

Dataset size: We evaluate the effectiveness of our proposed approach in zero-shot transfer tasks by varying both the dataset size and model depth. For dataset size, we assess model improvement over random prediction results by randomly sampling datasets of different sizes: 30,000, 60,000, 120,000, 240,000, and the full training set size of 456,360. For each of these training set sizes, we conduct two experiments. The first experiment uses a testing set of 50 randomly selected samples, while the second experiment employs testing sets of varying sizes: 50, 100, 300, 600, and 1,000 randomly selected samples.

Baselines: *CLCP*$_{lp}$: CLCP with local pooling layers. For the different sizes of training/testing sets, this type of CLCP is divided into five subtypes: CLCP$_{lp}(3) \sim$ CLCP$_{lp}(7)$, representing CLCP$_{lp}$ with 3, 4, 5, 6 and 7 blocks. A block indicates a functional unit containing a convolution layer and a pooling layer. *CLCP*$_{gp}$: CLCP also uses global pooling layers. This type of CLCP is divided into five subtypes: CLCP$_{gp}(3) \sim$ and CLCP$_{gp}(7)$. *CLCP*$_{rn}$: Inspired by ResNet's success in image coding and its use in CLIP (He et al., 2016), we adapt its architecture for our code encoder by replacing 2D convolution with 1D convolution. The model consists of an input layer, N residual blocks, and a global pooling layer. Each block includes three convolutional layers: the first two process input data sequentially, while the third performs a 1×1 convolution on the input directly. The output of the third layer is added to the second layer's output, forming the residual block's final output. For different dataset sizes, N varies as $3 \sim 7$.

Evaluation Metrics: *Accuracy (Acc.):* The proportion of the correctly matched (code, text) pairs in the L testing pairs. If all L codes are randomly matched to the texts, the estimated accuracy (EA) is $1/L$.

4.3.2 Settings for RQ2

Dataset size: We use the same dataset sizes in RQ1.

Baselines: Two of the most effective baselines in RQ1 are used, CLCP$_{lp}$ and CLCP$_{rn}$ with different blocks. Their variants are also used as baselines: *CLCP*$_{lp}$*+BN*: A variant of CLCP$_{lp}$ where the input of each convolutional layer is normalized by BN. *CLCP*$_{lp}$*-Pool*: A variant of CLCP$_{lp}$ where the pooling layer is removed. *CLCP*$_{lp}$*-Init*: A variant of CLCP$_{lp}$ where the He initialization is removed. *CLCP*$_{rn}$*+BN*: A variant of CLCP$_{rn}$ where the input of each convolutional layer is normalized by BN. *CLCP*$_{rn}$*-Pool*: A variant of CLCP$_{rn}$ where the pooling layer is removed. *CLCP*$_{rn}$*-Init*: A variant of CLCP$_{rn}$ where the He initialization is removed.

Evaluation Metrics: In this RQ, we calculate the difference between the average accuracy of CLCP$_{lp}$/CLCP$_{rn}$ with different blocks and the average accuracy of their variants across all sizes of datasets to determine the variation in the effect of the variants relative to the original model.

4.4 Experiment Results

Answer to RQ1: As shown in Fig. 9, the legend indicates model and training set sizes, with a fixed testing set of 50 samples. The colors represent model accuracy growth across different sizes. As both the training set and model size increase, all models improve in performance on the testing sets and surpass random prediction accuracy (EA). This

[2]https://opendatalab.org.cn/CodeSearchNet

suggests CLCP's effectiveness in learning. Larger data amounts could further enhance performance. Notably, variants using local pooling outperform those using global pooling, possibly due to global pooling's loss of local information critical for tasks, while local pooling retains more local details beneficial for fine feature representation in code images.

Regarding Fig. 10, the legend similarly represents model and training set sizes, with the testing set size increasing proportionally (50, 100, 300, 600, 1000 samples). As both the test set and model size grow with the training set, model performance declines, indicating underfitting. This may stem from insufficient training data compared to CLIP's 400 million points. Without NLP embedding techniques, our redefined code representation paradigm increases data diversity and complexity, making it challenging for the model to learn general patterns from limited data.

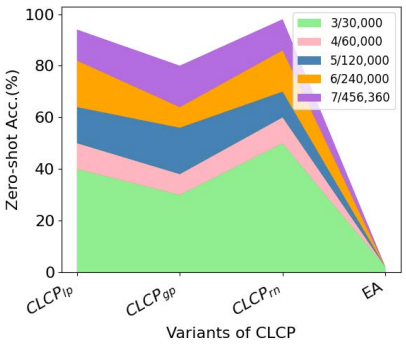

Figure 9: As the sizes of the training set and model increase, the performance of variants on the testing sets of same size.

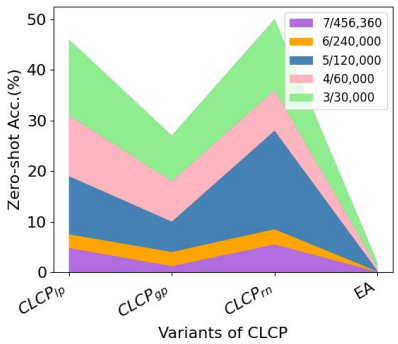

Figure 10: As the sizes of the training set and model increase, the performance of variants on the testing sets of different size.

Answer to RQ2: Fig. 11 shows the accuracy drop of CLCP$_{lp}$ and CLCP$_{rn}$ models when components are added or removed, compared to the original model. EA represents the baseline random prediction accuracy. The vertical axis indicates component changes (+/-), and the horizontal axis shows the average accuracy decrease across different model and dataset sizes (same as RQ1). Legends denote different original models.

Removing the pooling layer caused a significant accuracy drop, with *CLCP$_{lp}$-Pool* performing worse than random prediction. Training validation loss for *CLCP$_{lp}$-Pool* and *CLCP$_{rn}$-Pool* stalled after 2-3 epochs, indicating overfitting and local optima. Removing He initialization also reduced performance for *CLCP$_{lp}$-Init* and *CLCP$_{rn}$-Init*. Adding Batch Normalization (BN) decreased performance, likely because BN disrupts relative differences between code feature maps by normalizing each batch independently.

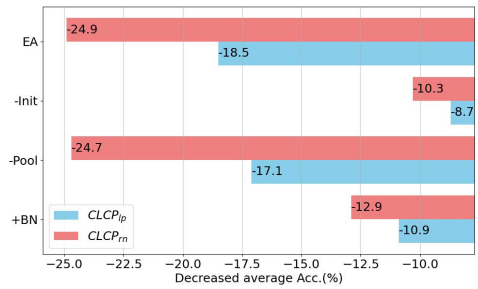

Figure 11: Removing and adding different components, changes in the effectiveness of CLCP$_{lp}$ and CLCP$_{rn}$.

Conclusion

Microsoft Research and Google DeepMind identified GPT's autoregressive limitations, including lack of planning, memory, backtracking, and reasoning. GPTs, relying on local, greedy word generation, struggle with complex logic and unseen code, heavily influenced by prompt format. To address this, we propose a new code encoding paradigm inspired by diffusion techniques in image generation, encoding code as heterogeneous images with global information. We designed a text-to-code encoder model based on Sora's CLIP, learning global code understanding and connecting text-code encoding spaces. Through self-supervised learning on 456,360 pairs, the model achieved zero-shot prediction, paving the way for diffusion-based code generation to overcome autoregressive limitations.

Limitations

Due to hardware and data limits, we couldn't train CLCP models with as many datasets as OpenAI

used for CLIP. We proposed text-to-code based on CLIP's text-to-image (initially tested on limited data), resulting in a non-robust CLCP model unsuitable for downstream code tasks. This paper aims to offer a new perspective for code understanding researchers. Future plans include collecting more datasets, seeking funding for better hardware, and not comparing with pre-trained embedding works due to our new encoding paradigm. We replicated and improved CLIP's framework ourselves due to lack of openness, impacting model performance.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. 2024. Premise order matters in reasoning with large language models. *CoRR*, abs/2402.08939.

Karan Desai and Justin Johnson. 2021. Virtex: Learning visual representations from textual annotations. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 11162–11173. Computer Vision Foundation / IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1026–1034. IEEE Computer Society.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Ronghang Hu, Amanpreet Singh, Trevor Darrell, and Marcus Rohrbach. 2020. Iterative answer prediction with pointer-augmented multimodal transformers for textvqa. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 9989–9999. Computer Vision Foundation / IEEE.

Jack Dunger Josh Abramson, Jonas Adler. 2024. Accurate structure prediction of biomolecular interactions with alphafold 3. *Nat.*, pages 1476–4687.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324.

Ang Li, Allan Jabri, Armand Joulin, and Laurens van der Maaten. 2017. Learning visual n-grams from web data. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 4193–4202. IEEE Computer Society.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

William Peebles and Saining Xie. 2023. Scalable diffusion models with transformers. In *IEEE/CVF International Conference on Computer Vision, ICCV 2023, Paris, France, October 1-6, 2023*, pages 4172–4182. IEEE.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.

Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with CLIP latents. *CoRR*, abs/2204.06125.

Mert Bülent Sariyildiz, Julien Perez, and Diane Larlus. 2020. Learning visual representations with caption annotations. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part VIII*, volume 12353 of *Lecture Notes in Computer Science*, pages 153–170. Springer.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nat.*, 625(7995):476–482.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Yuhao Zhang, Hang Jiang, Yasuhide Miura, Christopher D. Manning, and Curtis P. Langlotz. 2022a. Contrastive learning of medical visual representations from paired images and text. In *Proceedings of the Machine Learning for Healthcare Conference, MLHC 2022, 5-6 August 2022, Durham, NC, USA*, volume 182 of *Proceedings of Machine Learning Research*, pages 2–25. PMLR.

Yuhao Zhang, Hang Jiang, Yasuhide Miura, Christopher D. Manning, and Curtis P. Langlotz. 2022b. Contrastive learning of medical visual representations from paired images and text. In *Proceedings of the Machine Learning for Healthcare Conference, MLHC 2022, 5-6 August 2022, Durham, NC, USA*, volume 182 of *Proceedings of Machine Learning Research*, pages 2–25. PMLR.