EyeTrans: Merging Human and Machine Attention for Neural Code Summarization

YIFAN ZHANG, Vanderbilt University, USA
JILIANG LI, Vanderbilt University, USA
ZACHARY KARAS, Vanderbilt University, USA
AAKASH BANSAL, University of Notre Dame, USA
TOBY JIA-JUN LI, University of Notre Dame, USA
COLLIN MCMILLAN, University of Notre Dame, USA
KEVIN LEACH, Vanderbilt University, USA
YU HUANG, Vanderbilt University, USA

Neural code summarization leverages deep learning models to automatically generate brief natural language summaries of code snippets. The development of Transformer models has led to extensive use of attention during model design. While existing work has primarily and almost exclusively focused on static properties of source code and related structural representations like the Abstract Syntax Tree (AST), few studies have considered human attention — that is, where programmers focus while examining and comprehending code. In this paper, we develop a method for incorporating human attention into machine attention to enhance neural code summarization. To facilitate this incorporation and vindicate this hypothesis, we introduce EyeTrans, which consists of three steps: (1) we conduct an extensive eye-tracking human study to collect and pre-analyze data for model training, (2) we devise a data-centric approach to integrate human attention with machine attention in the Transformer architecture, and (3) we conduct comprehensive experiments on two code summarization tasks to demonstrate the effectiveness of incorporating human attention into Transformers. Integrating human attention leads to an improvement of up to 29.91% in Functional Summarization and up to 6.39% in General Code Summarization performance, demonstrating the substantial benefits of this combination. We further explore performance in terms of robustness and efficiency by creating challenging summarization scenarios in which EyeTrans exhibits interesting properties. We also visualize the attention map to depict the simplifying effect of machine attention in the Transformer by incorporating human attention. This work has the potential to propel AI research in software engineering by introducing more human-centered approaches and data.

CCS Concepts: • **Software and its engineering** → **Software creation and management**; • **Computing methodologies** → **Artificial intelligence**; • **Human-centered computing** → *Interaction design.*

Additional Key Words and Phrases: Eye-tracking, Human Attention, Machine Attention, Transformer, Code Summarization

Authors' addresses: Yifan Zhang, Vanderbilt University, USA, yifan.zhang.2@vanderbilt.edu; Jiliang Li, Vanderbilt University, USA, jiliang.li@vanderbilt.edu; Zachary Karas, Vanderbilt University, USA, z.karas@vanderbilt.edu; Aakash Bansal, University of Notre Dame, USA, abansal1@nd.edu; Toby Jia-Jun Li, University of Notre Dame, USA, toby.j.li@nd.edu; Collin McMillan, University of Notre Dame, USA, cmc@nd.edu; Kevin Leach, Vanderbilt University, USA, kevin.leach@vanderbilt.edu; Yu Huang, Vanderbilt University, USA, yu.huang@Vanderbilt.Edu.

ACM 2994-970X/2024/7-ART6
https://doi.org/10.1145/3643732

ACM Reference Format:
Yifan Zhang, Jiliang Li, Zachary Karas, Aakash Bansal, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. 2024. EyeTrans: Merging Human and Machine Attention for Neural Code Summarization. *Proc. ACM Softw. Eng.* 1, FSE, Article 6 (July 2024), 22 pages. https://doi.org/10.1145/3643732

1 INTRODUCTION

A code summary is a brief description representing the purpose and function of code that can aid developer comprehension [61]. Code summaries are a common part of documenting source code — while typically provided by the developer, machine learning models have increasingly been used to automatically generate summaries to augment documentation and improve comprehension. When documenting or summarizing source code, programmers' attention will focus on different parts of the code to formulate a general idea about it, eventually establishing a comprehensive understanding [24].

Several studies have developed an initial understanding of how human developers comprehend code. To capture the nuanced attention shifts of programmers reading code, a popular approach is to conduct eye-tracking studies [2, 50] to empirically analyze common eye gaze patterns during the process. Intuitively, certain patterns of eye gaze when examining source code correspond to a change in attention within the code [45, 63], which is usually highly correlated with changes in human concentration and is potentially valuable for gaining a better understanding of the cognitive processes in programming [4, 10, 27, 55]. Researchers have conducted several eye-tracking studies to explore human attention during code summarization. These usually involve university students or professional developers drafting or evaluating summaries for code snippets and recording their eye gaze patterns to understand human attention in such activities [49, 51]. Though the focus has been primarily on understanding the eye-tracking patterns, these studies have unveiled certain correlations between human visual attention and code summarization [2, 50], paving the way for future work to improve relevant tool design leveraging human attention.

In recent years, in tandem with the traditional definition of "attention" in human activities, *attention* can also refer to a concept in machine learning that describes the importance of weights in the layers of neural networks [23, 42]. Within this area, the *Transformer* [67] architecture is one of the most influential baseline structures that incorporate *neural attention* into a machine learning model. With advancements in machine attention mechanisms for system applications [68, 76, 77], an increasing number of Transformer-based models have been developed to address a variety of neural code summarization tasks [5, 21, 64, 65]. These methods leverage specific domain knowledge to modify their model structures accordingly, incorporating elements such as the Abstract Syntax Tree (AST) [65], dataflow graph [21], and function call graph [35]. To generate code summaries, these Transformer models essentially aim to "comprehend" the code, which is a task that human programmers already accomplish. It is therefore possible that human attention can be leveraged to improve these models' performance [52], which, to the best of our knowledge, has not yet been attempted. This process can be framed into the following questions: can human attention be integrated into complex machine attention mechanisms to enhance overall performance? If so, how can we pragmatically incorporate human attention into attention-based machine learning models? These questions are particularly relevant for Software Engineering (SE) and AI communities considering the Transformer architecture, which underlies the most recent advanced AI applications in SE, and large language models (LLMs) in natural language processing (NLP) [22, 40, 66].

In this paper, we hypothesize that human attention can enhance Transformer models, and with the understanding that a formalized methodology for this integration has yet to be explored, we introduce EyeTrans. EyeTrans leverages human eye-tracking data during training to effectively integrate human attention with machine attention, as illustrated in Figure 1. To this end, we first

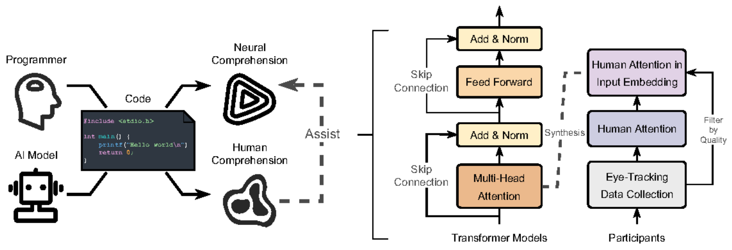


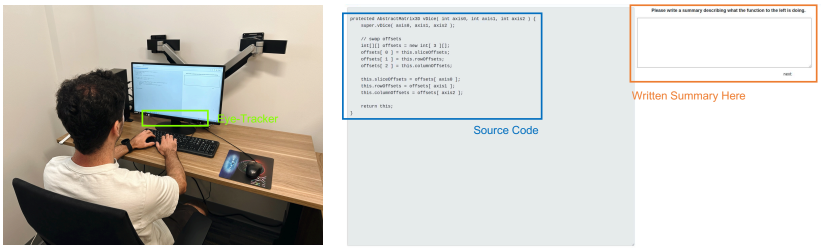
Fig. 1. Conceptual Overview of EyeTrans. EyeTrans synthesizes human attention information into the self-attention mechanism of Transformers, assisting Transformers in code summarization tasks.

conduct a human study to collect eye-tracking data from 27 programmers, which we then preprocess and incorporate into the Transformer. We describe our methodology, then present evaluation metrics that assess EyeTrans's performance. Specifically, we employ two code summarization-based evaluation methods to illustrate the effectiveness of human attention in augmenting neural code summarization: (1) classification-based code summarization, referred to as *Functional Summarization*, in which a model predicts the name of function (e.g., "open" or "sort") given its source code. This serves as a foundational step, enabling programmers to consolidate a preliminary understanding of the inherent functionality and semantics of the code. (2) Sequence-to-sequence (seq2seq) based code summarization, referred to as *General Code Summarization*, in which a model generates a sequence of natural language tokens succinctly describing what a given function does (e.g., documentation for the function). We find that integrating human attention results in a performance improvement of up to 29.91% in functional summarization, and up to 6.39% in general code summarization, thereby demonstrating the effectiveness of EyeTrans. We view our work as a proof-of-concept and hope our work can inspire more future work to leverage human factor research in AI for software engineering. We included the eye-tracking data collected in the study (de-identified) and the experimental code in the supplementary package and plan to release them to the public.

The contributions of this paper are as follows:

- We propose EyeTrans, the first approach to demonstrate that integrating human attention with the Transformer architecture can substantially enhance the performance of neural code summarization.
- We conduct exploratory examinations of eye-tracking data and rationalize the integration of human attention through preliminary analysis.
- We design a data-centric method to incorporate human attention into Transformers without altering its structure, and present comprehensive quantitative analyses to validate the performance of our approach.
- We illustrate alterations in the Transformer's attention maps to qualitatively demonstrate the changes brought by combining human attention and attention layer in Transformer.
- We review existing attention synthesis methods in relation to our work and outline prospective avenues for future research.

Paper Organization: Section 2 presents the design for the eye-tracking human study in this paper, and provides background knowledge on EyeTrans and the Transformer architecture. Section 3 elaborates on the details of EyeTrans. Section 4 details the experimental setup. Section 5 analyzes the results. Section 6 discusses threats to validity. Section 7 surveys related work. Finally, Section 9 concludes the paper and outlines future research directions.


Eye-Tracker
Source Code
Please write a summary describing what the function to the left is doing.
next
Written Summary Here

```
protected AbstractMatrix3D vDice( int axis0, int axis1, int axis2 ) {
    super.vDice( axis0, axis1, axis2 );

    // swap offsets
    int[][] offsets = new int[ 3 ][];
    offsets[ 0 ] = this.sliceOffsets;
    offsets[ 1 ] = this.rowOffsets;
    offsets[ 2 ] = this.columnOffsets;

    this.sliceOffsets = offsets[ axis0 ];
    this.rowOffsets = offsets[ axis1 ];
    this.columnOffsets = offsets[ axis2 ];

    return this;
}
```

Fig. 2. (*Left*) The experimental room, with the task displayed on the monitor. The Tobii Pro Fusion Eye-tracker is a thin bar magnetized to a strip at the bottom of the monitor. (*Right*) A screenshot of one task example, with the Java method displayed on the left of the screen, and the summary writing location in the top right.

2 PRELIMINARIES

The high-level idea of EyeTrans is to integrate human attention, collected from eye-tracking, into Transformer models to enhance the performance on code summarization. In this section, we first introduce the design for the human eye-tracking study, where the **eye-tracking data** is collected and used in EyeTrans, and then provide relevant background information for the **Transformer** architecture, especially the self-attention module of the Transformer model.

2.1 Eye-Tracking Study Design

We first conducted an eye-tracking study with 29 student programmers where participants completed Java source code summarization tasks. In this subsection, we describe details of the task design and experiment protocol of this human study.

Java Methods and Task Design We constructed a dataset of 162 Java methods for summarization tasks in our study using the FunCom dataset [37]. These methods were randomly selected from FunCom, then filtered based on their length to fit on the monitor without scrolling[1]. In this final dataset of 162 Java methods, the shortest contained 5 lines of code, while the longest contained 26 (μ=11.72, σ=4.25). The most complicated method had a cyclomatic complexity of 11 and the simplest had a complexity of 1 (μ=2.59, σ=1.56). In the experiment, methods were presented one at a time, along with an empty text box for participants to type their summaries, as shown in Figure 2. The methods were presented on a 24" monitor, without syntax highlighting and following standard Java formatting [41]. Participants' eye gaze data was recorded during the tasks using the the Tobii Pro Fusion eye-tracker (60Hz), which is accurate to 0.1–0.2 inches (0.26–0.53cm) on the monitor [1].

Recruitment and Experimental Protocol We recruited 29 undergraduate and graduate CS students in our study with IRB approval at *elided for double blind*. All participants had taken the Data Structures course or equivalent and passed a Java coding test during pre-screening. All participants who completed the study were compensated $60. Due to a protocol error in one case and software malfunction in another, 27 of the 29 participants' data were included in the final dataset. These 27 participants were 23.8 years old on average, and include 8 women and 15 graduate students.

The experiment was conducted in person, in an office with natural lighting (as shown in Figure 2). During the entire experiment, participants completed 24 or 25 *stimuli*, where one stimulus consisted of a Java method and participants were asked to type their summaries for the method. A break period was built halfway into the task interface, both for participants to rest and for the researcher to recalibrate the eye-tracker (for data quality). Each participant took roughly 50 minutes to complete the entire experiment.

[1]A static screen of code that does not scroll facilitates the collection of eye tracking data.

2.2 Self-Attention in Transformers

The Transformer architecture [67] is a seminal model in NLP and the backbone of EyeTrans. The architecture is highly reliant the *self-attention mechanism*, which interrelates tokens in the input sequence with surrounding context. In this section, we highlight how self-attention functions within the Transformer. Later, in Section 3.3, we discuss how we augment model self-attention with human attention to improve the Transformer's performance.

Before input sequences are fed into the Transformer, each token in the sequence is mapped into an embedding space, where it is represented as a vector. Conventionally, a token's input embedding is derived from the summation of its semantic and positional embeddings. In EyeTrans, we introduce a third embedding component. This additional embedding term relates pairs of tokens examined by human programmers in close temporal proximity, thus capturing human visual attention patterns during code comprehension.

In Transformer architectures, *self-attention* helps the model learn how related two tokens' embedding representations are. Informally, the more related two tokens' embedding representations are, the more a self-attention layer learns to associate these tokens together. As such, Transformers' attention largely refers to an ability to learn the inter-relation between tokens. Consequently, we intuit that we can improve this self-attention mechanism by incorporating human attention based on the program elements humans consider during comprehension [4, 6]. *We hypothesize that a human's way of associating elements in code may capture subtle inter-relations that are not easily learned by the Transformer.*

More formally, the self-attention mechanism transforms each token's embedding into three vectors: query (Q), key (K), and value (V). Conceptually, the key/value/query concept is analogous to a search engine. Each input token, denoted $token_0$, aims to find other related tokens. To accomplish this, $token_0$ uses its query vector, Q_0, to identify related tokens, similar to a search query into an indexed database. In response to this query vector, every other input token, denoted $token_i$, provides a key vector, K_i. If Q_0 closely matches some K_i, then the mechanism deems $token_i$ relevant to $token_0$, and presents $token_i$'s content in response, represented by its value vector, V_i. This vector V_i is then mixed with $token_0$'s value vector V_0.

Given input embedding matrix X, the matrices for query, key, and value are $Q = XW_q, K = XW_k, V = XW_v$, where W_q, W_k, W_v are learned parameters. Self-attention is then calculated as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V, \text{where } d_k \text{ is the dimension of key vector.} \quad (1)$$

The QK^T term in this equation is intuitively analogous to a similarity matrix representing pair-wise relatedness between tokens.

Equation (1) describes a single instance of attention, but in practice, multiple attention functions are performed in parallel, leading to multi-head attention. While attention is the essence of the Transformer, the architecture has other components that contribute to its effectiveness. As illustrated in Figure 1, the multi-head attention's output is subject to layer normalization, fed through a fully connected network, and normalized again. This sequence of layers, starting from multi-head attention to layer normalization, constitutes one *Transformer block*. When multiple blocks are connected in sequence, the output from one block serves as the input for the next, and the final output can be used for a variety of downstream tasks. In EyeTrans, we employ stacks of Transformer blocks as the NLP model for code summarization, as discussed later in Section 4.2.

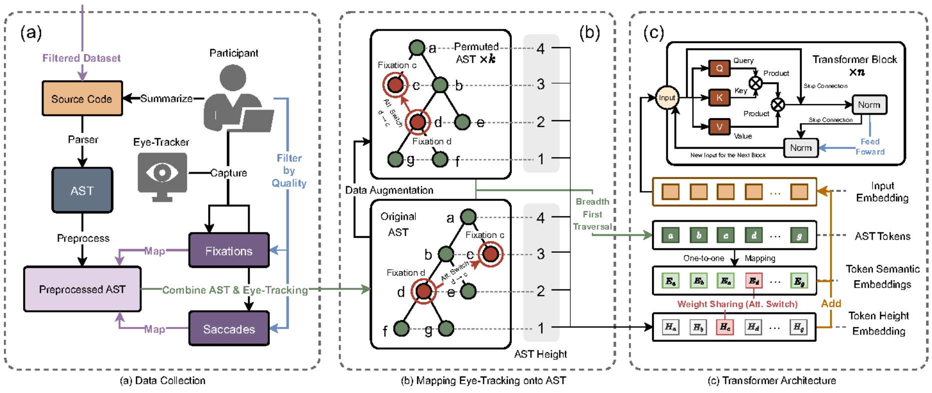


Fig. 3. EyeTrans Methodology Overview. (a) depicts our data collection processes. (b) describes the data preprocessing steps, including transformation to ASTs, data augmentation, and representing attention switches as edges on ASTs. (c) underlines our proposed approach to model attention switches using Transformers' default input embedding modality.

3 APPROACH

In this section, we introduce the methodology in EyeTrans to combine machine and human attention to enhance neural code summarization. We integrate human attention into EyeTrans by implanting eye-tracking data into the Transformer's input embeddings. Our approach for combining machine and human attention consists of the following steps: we (1) preprocess source code and (2) preprocess eye-tracking data to obtain preliminary datasets. Then, we (3) map the eye-tracking data onto the source code by combining their respective embedding representations. In doing so, we can improve Transformer-based summarization by leveraging how humans examine and comprehend code when summarizing it.

3.1 Source Code Pre-Processing

To preprocess the source code, we first use the srcML parser to convert the textual format of each Java method into a corresponding AST representation [17]. Transforming the code into ASTs is suitable for neural code summarization because this representation provides domain-specific, structural information of source code [7, 20, 21, 26, 36, 38, 59, 65]. This structural information is encoded within the tree structure of nodes, where each node describes both token types and token literals. For instance, each of the three nodes in {`'i'`, `'<'`, `'0'`} may be a child node to a `conditional statement` node. The ASTs serve two purposes in this study: (1) to provide a suitable, structured representation for training a Transformer-based model, and (2) to facilitate mapping eye tracking data to corresponding program structures in a snippet of source code.

Training Transformers on Code Structure We first parse source code into corresponding ASTs, and then discard token literals to abstract away individual program details. For example, a conditional expression such as `i < 0` may instead be represented as *variable* < *integer* to avoid overfitting to specific token values while retaining appropriate structural program information. This approach aligns with the code structural learning settings investigated and validated by CodeNN [29], emphasizing learning through code tokens devoid of literals.

We further employ ***data augmentation*** to grow the dataset. Typical human eye-tracking studies do not yield the volume of data required for modern machine learning. Thus we develop the augmentation method below to overcome this disadvantage. Given the original AST of a

Java method, we generate multiple variants of the AST (i.e., we augment the AST) via subtree permutations. Recursively, for each node within the AST, we randomly permute all subtrees of the node, altering the sequence of AST subtrees but preserving the hierarchical order (i.e., the parent-child relationship). For instance, in Figure 3 (b), the augmentation alters the order of nodes 'b' and 'c', but 'a' remains the parent of 'b' and 'c'. Although such permutations introduce changes to the semantics of the original program, they still preserve much of the core information of the original source code. For example, switching the order of two parallel assignment statements in a Java method generally retains the overall semantics. We empirically verify that the benefits of growing the dataset through augmentation outweigh the semantic deviations induced by AST permutations.

Analyzing Eye-tracking Data using Code Structure To analyze the eye-tracking data, we use the AST information to examine programmers' code reading patterns as they summarize code. Specifically, we sought to determine whether there exists consistent code-reading strategies that programmers used during the task. Informally, if programmers looked equally at every token literal in a snippet of code, it may not be meaningful to use eye-tracking data to assist neural code summarization. Thus, within the eye tracking data, we consider the most common transitions where the eye focuses within snippets of source code to represent programmers' attention. For instance, if a programmer looks at a method declaration, the *next* element they look at could be the return statement, other method calls, or the arguments of that method. Here, we map the eye-tracking data back to *semantic categories* of structural program elements in the AST.

To determine which semantic categories to consider, we referred to a widely-used Java textbook [53]. The list of semantic categories includes the method declaration, variable declarations, return statements, conditional statements, and conditional blocks, among others. The complete list of 19 semantic categories can be found in the Supplementary Materials. Multiple labels may apply to a single token literal. In these scenarios, we assigned the most meaningful labels, based on the semantics of Java, the structural context of the literal, and prior code summarization and comprehension research [12, 13, 51]. For example, the equal sign is an operator, but can also belong to a variable declaration. In this case, we would label the equal sign as a 'variable declaration,' because this label conveys more substantial information than 'operator' in this context [15]. In brief, we extract the structural information of a program into an AST, then analyze the reading patterns of programmers as they comprehend code, and then map this information back to the AST in terms of high-level semantic categories.

3.2 Eye-Tracking Data and Preprocessing

In essence, raw eye-tracking data consists of screen coordinates and their timestamps. To map these gaze coordinates to token literals on the screen, we calculated bounding boxes around each token using the `opencv-python` library for computer vision. Using these pixel-coordinate boundaries, we then localized gaze coordinates in the eye-tracking data to corresponding bounding boxes.

Localizing gaze to token literals is a crucial preliminary step, but further processing is needed to make inferences about human cognition. Specifically, we extract characteristic visual patterns in humans' gaze behaviors, *fixations* and *saccades*, from the raw eye-tracking data [57]. A fixation is a spatially-stable eye gaze which typically lasts for 200–300 milliseconds. Most cognitive processing of visual information occurs during fixations [56]. Saccades are shorter in duration (40–50ms), and occur when humans make jumps in their visual field. In contrast to fixations, there is little cognitive processing that occurs during saccades [30]. Therefore, to understand patterns of human cognition, we needed to differentiate fixations from saccades [57]. By current standards, this distinction is made using the *velocity* of the eye-movement. In other words, if the speed of an eye-movement exceeds a certain threshold, a Velocity-Threshold Identification (I-VT) algorithm will identify it as a saccade [44]. Based on our implementation and following best practices [14], we considered an eye-movement as a saccade if it exceeded 400px/100ms.

Once fixations are identified in the eye-tracking data, researchers typically use the number of fixations (i.e., *fixation count*), and their duration (i.e., *fixation duration*) as a proxy to measure cognitive processing [56]. For instance, a higher fixation count and longer fixation durations signify greater cognitive effort [30]. By using fixations as building blocks, researchers can glean more complex cognitive behaviors from eye-tracking data. For instance, researchers can examine the *ordered sequences* of fixations (i.e., scan paths) for insights into the flow of human attention [57]. In this paper, we use **attention switch** to refer to transitions between fixations within these ordered sequences (i.e., a saccade connecting two distant fixation points). As such, each programmer's eye-tracking data on each Java method can be seen as a sequence of attention switches.

Each attention switch can also be represented on the AST as a directed edge from one node to another, as depicted in Figure 3 (b). For each AST in our dataset, we therefore have derived attention switch edges from the human eye-tracking data. When we augment the original ASTs via permutation to generate variants, we also map the attention switch edges onto the permuted ASTs, preserving the endpoints of each attention switch. For example, in Figure 3 (b), an attention switch edge connecting nodes 'd' an 'c' in the original AST would still connect these two nodes in the permuted AST. Thus, every permuted AST possesses an enduring set of attention switch edges, which ensures the integrity of the attentional data amidst structural alterations.

3.3 Modeling Human Attention into Transformers

In this section, we introduce our approach to model human attention into Transformer. Our key idea centers around mapping *attention switches*, defined in Section 3.2, onto each of the corresponding augmented ASTs generated in Section 3.1.

Our approach is inspired by the following intuition: attention switches between two token literals may represent a synthesis of information between them [4, 6], which may also signify a relation between these tokens during human code comprehension. Therefore, we aim to incentivize the self-attention mechanism within Transformers to discern the implicit relatedness that attention switches reveal between input tokens. As we discuss in Section 2.2, the self-attention naturally associates tokens with similar embeddings. Thus, we facilitate the recognition of relatedness between token pairs that are connected by an attention switch by introducing a shared embedding component for such pairs. Following this intuition, the detailed process of modeling human attention into Transformers can be divided into two steps: (1) mapping AST tokens and attention switches onto distinct embedding spaces and (2) combining the distinct embeddings obtained in the previous step to create a single input embedding vector for each token.

Mapping Data to Three Embedding Spaces For each AST, we perform a breadth-first-search traversal starting at the root node to obtain a sequential representation of the AST, as illustrated in Figure 3 (c). We denote this sequence of tokens $\{a, b, c, \dots\}$. In standard Transformer architectures, each token is depicted by a semantic embedding and a positional embedding:

$$\text{Semantic Embedding } \{E_a, E_b, E_c, \dots\} := \{f_e(a), f_e(b), f_e(c), \dots\},$$
$$\text{Positional Embedding } \{H_a, H_b, H_c, \dots\} := \{f_h(h(a)), f_h(h(b)), f_h(h(c)), \dots\}.$$

Here, h is a function that returns the height of the input token on the AST, and f_e, f_h map to distinct embedding spaces. Intuitively, the semantic embedding vector captures the token's *type*, and the positional embedding vector represents the token's *height* on the AST.

After projecting AST tokens into these two embedding spaces, we proceed to convert attention switches into their own embedding representation. Each attention switch can be represented as a directed edge linking two AST nodes, as illustrated in Figure 3 (b). Formally, a sequence of attention

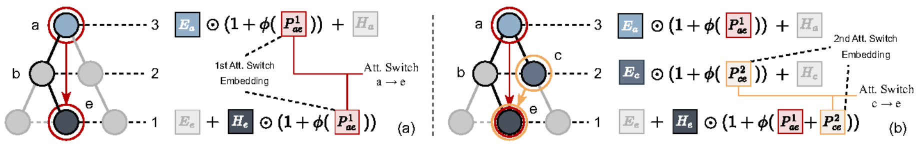


Fig. 4. Illustration of how EyeTrans models attention switch. Attention switches, conceptualized as edges connecting AST nodes, are mapped onto corresponding AST tokens by merging their respective embeddings.

switch edges on an AST is denoted as $\{s^1_{\alpha_1\beta_1}, s^2_{\alpha_2\beta_2}, s^3_{\alpha_3\beta_3}, \ldots\}$. The superscripts align the attention switch edges in temporal order, and $\alpha_i, \beta_i \in \{a, b, c, \ldots\}$ denote the origin and the target tokens of the directed edge. For example, s^2_{ce} (with superscript 2, $\alpha_2 = c, \beta_2 = e$) denotes a participant's second attention switch in a given AST, transitioning from token 'c' to 'e'. We use an embedding mapping f_p to obtain embedding representations of attention switch edges:

$$\text{Attention Switch Embedding } \{P^1_{\alpha_1\beta_1}, P^2_{\alpha_2\beta_2}, P^3_{\alpha_3\beta_3}, \ldots\} := \{f_p(s^1), f_p(s^2), f_p(s^3), \ldots\}$$

Note that f_e, f_h, f_p are distinct functions mapping to three separate embedding spaces. Furthermore, f_p disregards the origin and target tokens of an attention switch edge. Hence, attention switch edges across different Java methods possessing identical superscripts (temporal cardinality) receive the same embedding representation. For instance, if two attention switch edges in two different Java methods are both the earliest attention switch for their respective methods, they are assigned the same embedding vector. In doing so, we generalize attention switch embeddings across different methods and encode temporal ordinality information into attention switch embeddings.

Combining Three Embedding Spaces Following these steps, we obtain sequences of semantic and positional embeddings of AST tokens via BFS on the AST graph. Additionally, the attention switch embeddings were derived as edges on a graph representation of the AST. In this step, we map the attention switch edges onto corresponding AST tokens' embeddings, combining different embedding representations into a singular, sequential input embedding vector per AST token. We illustrate this mapping via two equivalent perspectives: explaining (i) what happens to each token and (ii) what happens to each attention switch edge.

Perspective (i): what happens to each AST token? We answer this question with formal equations. Given some AST node, denoted $\gamma \in \{a, b, c, \ldots\}$ with token encoding E_γ and positional encoding H_γ. We sum up the embeddings of all attention switch edges that originate from γ, denoted $\sum_{\alpha_k=\gamma} P^k_{\alpha_k\beta_k}$. Informally, this summation embodies all attention switches originating from token γ. We then map this sum onto the γ's semantic embedding E_γ through

$$E_\gamma \odot (1 + \phi(\sum_{\alpha_k=\gamma} P^k_{\alpha_k\beta_k})), \tag{2}$$

where $\odot$ is element-wise multiplication and ϕ is the *ReLU* function. Next, with a similar process, we sum up the attention switch embeddings of all attention switch edges that point to γ, denoted $\sum_{\beta_k=\gamma} P^k_{\alpha_k\beta_k}$. We then map this value onto the γ's positional encoding H_γ, obtaining

$$H_\gamma \odot (1 + \phi(\sum_{\beta_k=\gamma} P^k_{\alpha_k\beta_k})). \tag{3}$$

Finally, we sum up Expressions (2) and (3), which are γ's semantic and positional embeddings, both modified by attention switch embeddings, to obtain the final embedding representation of γ:

$$E_\gamma \odot (1 + \phi(\sum_{\alpha_k=\gamma} P^k_{\alpha_k\beta_k})) + H_\gamma \odot (1 + \phi(\sum_{\beta_k=\gamma} P^k_{\alpha_k\beta_k})) \tag{4}$$

For token γ, Expression (4) captures the token's semantic and positional embeddings, as well as accounting for all attention switch edges originating from and leading to token γ. Expression (4) is then used as token γ's final input embedding vector fed into the Transformer.

Perspective (ii): what happens to each attention switch edge? We answer this question through an exemplary case study, illustrated in Figure 4 (a). The attention switch edge described in the figure, being the first attention switch from a to e, can be denoted s^1_{ae} with embedding P^1_{ae}. We map this edge's embedding P^1_{ae} onto token a's semantic embedding E_a, obtaining $E_a \odot (1 + \phi(P^1_{ae}))$. Then, we map the same edge's embedding onto token e's positional embedding H_e, obtaining $H_e \odot (1 + \phi(P^1_{ae}))$. In this way, the direction of this edge is preserved: the mapping onto the *input* embedding indicates a is the origin of the attention switch edge, while the mapping onto *positional* embedding indicates e is the target of the edge.

Note that in Figure 4 (a), after we add up each token's semantic and positional embeddings, both modified by attention switches, the resulting embedding representations of token a and c become more related. This is because both token a and e's aggregated embedding vectors now share a same embedding component: P^1_{ae}. As illustrated in this example, our method introduces relatedness through a shared embedding component between token pairs connected by attention switch. Intuitively, such relatedness is recognizable by self-attention mechanisms due to the mechanism's proclivity to associate tokens with similar embeddings. Figure 4 (b) illustrates a similar process when a second attention switch edge is mapped onto the tokens, and this iterative process continues until all attention switches are incorporated.

Perspective (i) and (ii) are two ways of looking at the exact same procedure. Regardless of the perspective, the attention switch embeddings are mapped onto AST tokens' semantic and positional embeddings, resulting in a unified input embedding vector for each AST token. This unified embedding is then used as input for the Transformer-based models, enhancing their performance in code summarization tasks. Since the choice and details of Transformer models depend on the requirements of the downstream summarization tasks, we will introduce our Transformer architecture design in Section 4.

In essence, our approach allows us to implicitly represent directed human attention switch edges using Transformer's standard input embedding modality. Consequently, human attention switches during code comprehension establish connections between AST tokens in a format that is intuitively compatible and recognizable by Transformers' self-attention layers.

4 EXPERIMENTAL DESIGN

In this section, we present the experimental design for evaluating EyeTrans. Specifically, we detail (1) dataset preparation, (2) model design and choices of hyperparameters, (3) noise and dropout training designs to mitigate bias in the eye-tracking dataset, and (4) evaluation metrics. We then use this design to answer four Research Questions described in Section 5.

4.1 Dataset Preparation

We briefly introduce the preparation of our dataset, encompassing both the collection of the eye-tracking dataset and the subsequent tokenization of the train-test dataset.

Eye-Tracking Dataset We obtained the raw eye-tracking data from the human study as introduced in Section 2.1, along with the summaries participants wrote. An inaccurate summary indicates lesser comprehension of the code, so we excluded eye-tracking data associated with inadequate summaries. Specifically, we conducted manual filtering by two raters, Rater A and Rater B, to control for data quality. Each rater first assessed the written summaries independently using four metrics: Accuracy (a), Completeness (b), Verbosity (c), and English Proficiency (d), each rated from 1 to 5. Subsequently, the raters met in person to reconcile any discrepancies in their

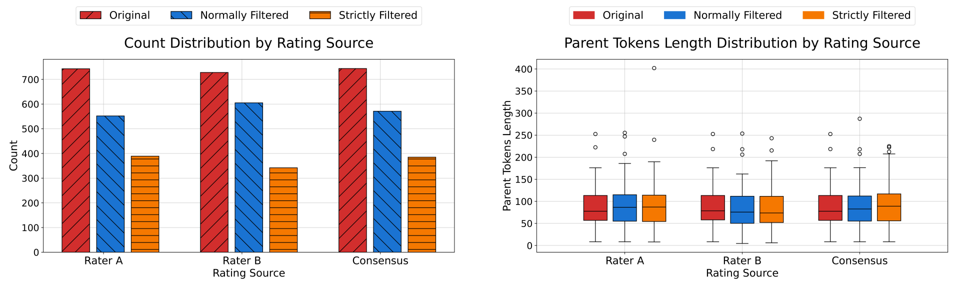


Fig. 5. Count and Parent Token Length Distribution by Rating Source. The count decreases for all rating sources as filtering tightens. The consensus approach mitigates outliers, which are seen in strictly filtered comments from Rater A.

scores and reach a consensus. As a result, we established three filtering standards based on these metrics. For the original standard (Original), we retained all valid data points without considering their scores; for the filtered dataset (Filtered), we included data with $a \geq 3, d \geq 3$ and $b \leq 3, d \leq 3$; for the strictly filtered dataset (Strict), we used data with $a > 3, d > 3$ and $b < 3, d < 3$. Figure 5 displays the count and parent token length distribution by rating source. Through filtering, we introduced three levels to data quality, thereby facilitating the analysis of how different levels of eye-tracking data quality impact the performance of Transformer models. We chose the consensus data for train-test data preparation.

Train-Test Dataset We adhere to the approach introduced in Sections 3.1 and 3.2 to tokenize the data and formulate the initial dataset. For the experiments, we eliminate all duplicate augmentations of ASTs to ensure that each combination of AST and eye-tracking data is unique. The final dataset comprises 10,676 original, 7,949 filtered, and 4,991 strictly filtered ASTs, each combined with eye-tracking data. For the Functional Summarization task, each augmented AST retains the original label of the source code. For the labels of the General Summarization task, we leverage an LLM [46] to paraphrase comments into semantically equivalent but varied comments to avoid reducing the task to mere memorization of comment sequences. We follow the typical approach of allocating 80% of the data to the training set and 20% to the test set, and set the random seed to 42 during dataset preparation. Before being fed into the Transformer models, initial AST and eye-tracking datasets are transformed into embedding representations, as described in Section 3.3.

4.2 Transformer Models and Hyper-Parameters

We now present our choices of Transformer models that consume the embedding representations of input datasets (obtained through methods described in Section 3) and perform the downstream summarization tasks. We detail the structures used for both Functional Summarization and General Code Summarization, as well as our choices of hyperparameters. For Functional Summarization, we implement 4 encoder layers and use the *CLS* token as the aggregated embedding for function classification. For General Code Summarization, we employ a seq2seq Transformer structure, composed of 4 encoder layers and 4 decoder layers. The embeddings from the decoder layers are used as inputs for Q and K after the encoder layers in the encoder-decoder attention mechanism. For both model types, we incorporate default residual connections [25] and layer normalization [8] in the feed-forward layers.

Hyper-Parameters For both the Functional Summarization and General Code Summarization models, we configure the total number of training epochs to 10, the learning rate to 1e-3, the hidden dimension to 32, and the number of attention heads to 4, employing Adam [32] with default settings as the optimizer. Unless specifically mentioned, we establish the maximum token of AST to 200, the maximum height to 50, the maximum number of classes to 300, the maximum number of vocab for comments to 2000, and the maximum length of summarization to 30. Additionally, we set the batch size to 256 during training and choose random seeds of 0, 1, 42, 123, and 12345 to repeat our experiments five times to obtain averaged results.

4.3 Dropout and Noise in Training Design

In addition to the standard hyperparameters, we introduce token-level dropout and Gaussian noise to simulate variations in data, thus creating increasingly challenging summarization tasks. Within such training scenarios, models need to not only perform well on the provided data but also be robust to in- and out-of-domain variations.

Robustness We evaluate the EyeTrans Transformer models' robustness by applying dropout to semantic token embeddings. To represent varying levels of adaptability, we define three dropout rates: 0.0, 0.1, and 0.5, denoted as R_0, R_1, and R_2 respectively. R_0 illustrates our model's performance under normal settings without dropout. We further assess model robustness by integrating Gaussian noise into semantic token embeddings. We introduce three levels of Gaussian noise: 0.0, 0.1, and 0.5, represented as N_0, N_1, and N_2 respectively, with N_0 demonstrating our model's performance in a standard setting without noise. Specifically, for RQ3, we employ an interval of 0.05 to generate a uniform mesh of noise levels between N_1 and N_2 for performance visualization.

4.4 Evaluation Metrics

We present our chosen quantitative evaluation metrics for both Ceneral and Functional Summarization, respectively.

For quantitative evaluation of Functional Summarization, we employ Mean Average F1 (MAF1), Mean Average Precision (MAP), and Mean Average Recall (MAR) to assess the model's performance in recovering the name of single functions. Besides these metrics, we conduct a case study to qualitatively evaluate how the introduction of human attention brings changes to Transformer's self-attention layers.

For quantitative evaluation of General Code Summarization, we use several ROUGE scores [39], comprising the unigram-based score, ROUGE-1; bigram-based scores, ROUGE-2, ROUGE-S, and ROUGE-SU; and the sentence-based score, ROUGE-L, to contrast the performance of EyeTrans and a baseline Transformer model. For each metric, we calculate the mean average to represent the general performance on single functions. We recognize that ROUGE scores might not be optimal [61], as there could be more apt metrics for code summarization [58]. Nonetheless, we opt for ROUGE because it aligns well with the intrinsic nature of learning-based models in varied domains and eases comparative analysis for researchers.

5 EXPERIMENTAL RESULTS

In this section, we answer four research questions based on our experimental design to demonstrate the effectiveness of integrating eye-tracking data in training Transformer models for neural code summarization:

- RQ1: What attention patterns do programmers exhibit as they read code? Are these patterns significant enough to be incorporated into machine learning models?
- RQ2: By integrating human and machine attention, to what extent can the performance of the Transformer model be enhanced?
- RQ3: How does incorporating human attention into the Transformer model improve learning efficiency and robustness?

- RQ4: What specifically changes in the self-attention layers of the Transformer when it is combined with human attention?

5.1 RQ1: Human Attention

To determine whether human attention is suitable for integrating with a Transformer model, we first needed to characterize programmers' gaze patterns. Informally, we sought to determine whether code reading patterns are consistent and meaningful across programmers. We first considered the amount of time participants spent reading the code, as well as their fixations, then calculated the most common attention switches (cf. Section 3.2).

We find that participants looked at each Java method for an average of 26.54 seconds (σ = 23.16s). Next, we find that participants averaged 94.92 fixation counts on each method (σ = 43.81 fixations). Here we consider the *average* fixation durations to describe the individual characteristics of each fixation, as opposed to a cumulative measure of fixation durations [57]. During the Java summarization tasks, programmers' average fixation durations were 0.114 seconds (σ = 0.037s). Based on the standard deviations in particular, it appears programmers show variety in their behavior on the task.

Code comprehension research suggests that programmers do not read code linearly, instead revisiting certain elements [16], and focusing their attention on a subset of the code [18]. To uncover whether predominant code reading patterns are present, we calculate the most common attention switches programmers made. Specifically, we first compiled the ordered sequences of programmers' fixations (i.e., scan paths). These lists provide an ordered record of the token literals that participants read. For instance, a programmer may read code in the following order: `String s`→ “`hello world`” → `return s`. This sequence is comprised of token literals, but in our analyses, we replaced literals with their semantic categories. The example above would become `variable declaration`→ `literal`→ `return`. Within these sequences of semantic categories, we then computed the most common pairs, such as `variable declaration` → `literal`, and `literal` → `return` from above. Within our dataset, we collected these ordered sequences of token literals for each participant, for every method they summarized, totaling 60,411 data points.

Calculating the most common pairs (i.e., attention switches) in this data, we find that programmers most frequently look from `method declaration`→ `variable declaration` (2,593). Next, we see the reverse: `variable declaration`→ `method declaration` (2,533). The third (2,189) and fourth (2,179) most common attention switches follow this reversing pattern, where programmers most commonly vacillate between the same two semantic categories: `conditional statement` ⇄ `loop body`. This persists for the fifth (1,615) and sixth (1,588) most common attention switches as well: `conditional statement` ⇄ `method declaration`. By analyzing the most common attention switches made by participants, we see structured code reading patterns emerge, where programmers vacillate between categories (i.e., `loop body` ⇄ `conditional statement`), and focus comparatively more on conditional statements. These results illustrate patterns in how humans read code, which may be beneficial for code comprehension. Thus, we next investigate the impact of integrating these attention patterns into Transformer models.

5.2 RQ2: Quantitative Analysis

Having conjectured the usefulness of our eye-tracking dataset, we next quantitatively analyze EyeTrans's demonstrated performance gain on neural code summarization tasks, focusing on both Functional Summarization and General Code Summarization tasks.

For Functional Summarization, Table 1 illustrates EyeTrans's substantially improved performance in MAF1, MAP, and MAR compared to the vanilla Transformer. We highlight the best performance across different scenarios and found that the improvement in MAF1 can be as high

Table 1. Comparison of EyeTrans against Transformer on Functional Summarization. In the table, R_1 and R_2 represent dropout rates of 0.1 and 0.5, respectively, while N_1 and N_2 represent Gaussian noise levels of 0.1 and 0.5, respectively. The average value of each data point was determined by running the experiments five times, using 0, 1, 42, 123, and 12345 as random seeds.

	MAF1@1				MAP@1				MAR@1			
Metrics	(R_1, N_1)	(R_2, N_1)	(R_1, N_2)	(R_2, N_2)	(R_1, N_1)	(R_2, N_1)	(R_1, N_2)	(R_2, N_2)	(R_1, N_1)	(R_2, N_1)	(R_1, N_2)	(R_2, N_2)
Transformer (Original)	96.90	64.62	90.47	49.70	96.53	61.90	88.46	46.85	97.74	71.29	92.90	57.74
EyeTrans (Original)	99.61	70.31	93.10	56.43	99.56	68.14	92.26	53.85	99.68	76.13	94.84	63.55
Improvement	+2.80%	+8.79%	+2.91%	**+13.52%**	+3.15%	+10.11%	+4.29%	**+14.95%**	+1.98%	+6.80%	+2.09%	**+10.06%**
Transformer (Filtered)	92.78	53.94	75.78	42.59	91.90	51.58	73.67	39.99	94.47	60.43	80.43	50.21
EyeTrans (Filtered)	96.09	58.44	89.74	54.40	95.61	56.01	88.74	51.95	97.02	65.11	91.92	61.28
Improvement	+3.56%	+8.35%	+18.42%	**+27.82%**	+4.03%	+8.59%	+20.51%	**+29.91%**	+2.71%	+7.73%	+14.33%	**+22.03%**
Transformer (Strict)	82.92	52.76	76.54	46.70	81.36	49.21	74.11	42.95	86.45	61.29	81.94	55.48
EyeTrans (Strict)	95.68	55.58	83.87	49.48	95.15	52.15	82.32	45.71	96.77	63.87	87.10	58.71
Improvement	**+15.38%**	+5.33%	+9.58%	+5.96%	**+16.94%**	+5.97%	+11.05%	+6.43%	**+11.95%**	+4.21%	+6.30%	+5.80%

Table 2. Comparison of EyeTrans against Transformer on General Code Summarization. In the table, R_0 and R_1 represent dropout rates of 0.0 and 0.1, respectively, while N_0 and N_1 represent Gaussian noise levels of 0.0 and 0.1, respectively. The average value of each data point was determined by running the experiments five times, using 0, 1, 42, 123, and 12345 as random seeds.

	ROUGE-1		ROUGE-2		ROUGE-S		ROUGE-SU		ROUGE-L	
Metrics	(R_0, N_0)	(R_1, N_1)	(R_0, N_0)	(R_1, N_1)	(R_0, N_0)	(R_1, N_1)	(R_0, N_0)	(R_1, N_1)	(R_0, N_0)	(R_1, N_1)
Transformer (Original)	71.55	64.89	49.30	40.29	45.35	36.51	52.63	44.39	69.52	62.75
EyeTrans (Original)	72.91	66.92	50.67	42.76	46.79	38.84	53.95	46.47	70.82	64.85
Improvement	+1.90%	**+3.13%**	+2.78%	**+6.12%**	+3.17%	**+6.39%**	+2.51%	**+4.70%**	+1.87%	**+3.35%**
Transformer (Filtered)	64.90	58.45	40.02	33.03	35.73	28.37	43.77	36.61	62.98	56.47
EyeTrans (Filtered)	66.89	60.18	42.01	34.44	37.71	29.79	45.63	38.01	64.91	58.16
Improvement	**+3.07%**	+2.96%	**+4.97%**	+4.27%	**+5.54%**	+5.00%	**+4.25%**	+3.82%	**+3.05%**	+3.00%
Transformer (Strict)	48.18	44.58	24.41	22.30	16.26	13.03	24.27	20.74	46.67	43.30
EyeTrans (Strict)	48.45	44.92	24.36	22.28	16.51	13.24	24.53	20.95	46.99	43.64
Improvement	+0.56%	**+0.76%**	−0.21%	−0.09%	+1.54%	**+1.61%**	**+1.07%**	+1.01%	+0.68%	**+0.79%**

as 29.91% (for MAP@1 under R_2 and N_2), showcasing promising potential. For the strictly filtered dataset, EyeTrans demonstrates improved performance on R_1 and N_1. However, this improvement declines in more challenging training scenarios with higher dropout and noise levels, indicating that training only with high-quality eye-tracking data may not be beneficial in all scenarios. For General Code Summarization, Table 2 shows a consistent improvement across all ROUGE metrics for EyeTrans, except on the Strict dataset where the performances of EyeTrans and Transformer are comparable. We highlight the best performance across different regular training settings and found that on the Original dataset, EyeTrans generally performs better when there are regularization terms (i.e., 6.39% for ROUGE-S under R_1 and N_1), while on the Filtered dataset, EyeTrans works better in a normal setting (i.e., 5.54% for ROUGE-S under R_0 and N_0).

5.3 RQ3: Robustness and Efficiency

Based on RQ2, we know that by combining human attention, the performance of the Transformer has improved for both tasks. We further investigate two key aspects: (1) whether the performance changes under challenging training scenarios with noise and dropout, and (2) how training efficiency changes with respect to the quality of human attention data.

This involves a detailed analysis of the robustness and training efficiency of EyeTrans with respect to data quality. To address (1), we plot the MAF1 curve for various combinations of human

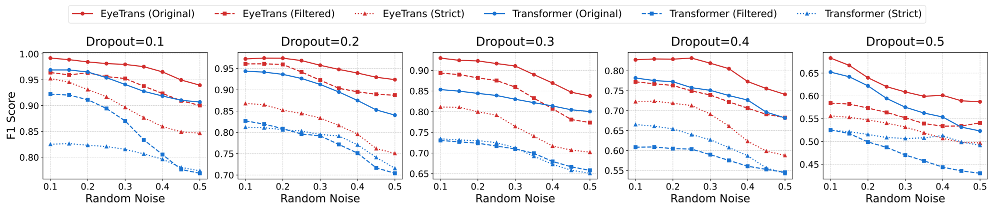


Fig. 6. Performance comparison of EyeTrans and Transformer on Functional Summarization with varying dropout and noise. The average MAF1 of each data point was determined by conducting the experiments five times, using 0, 1, 42, 123, and 12345 as random seeds.

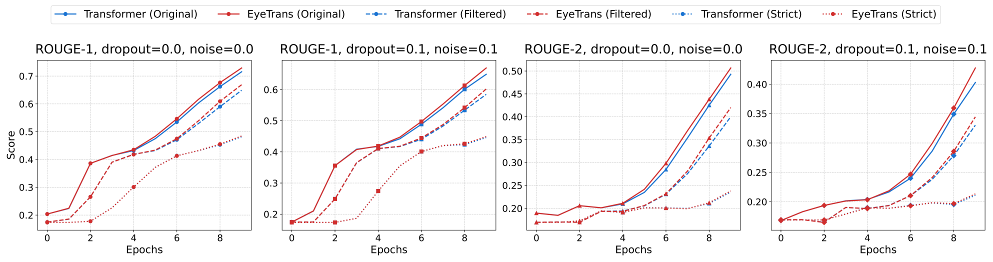


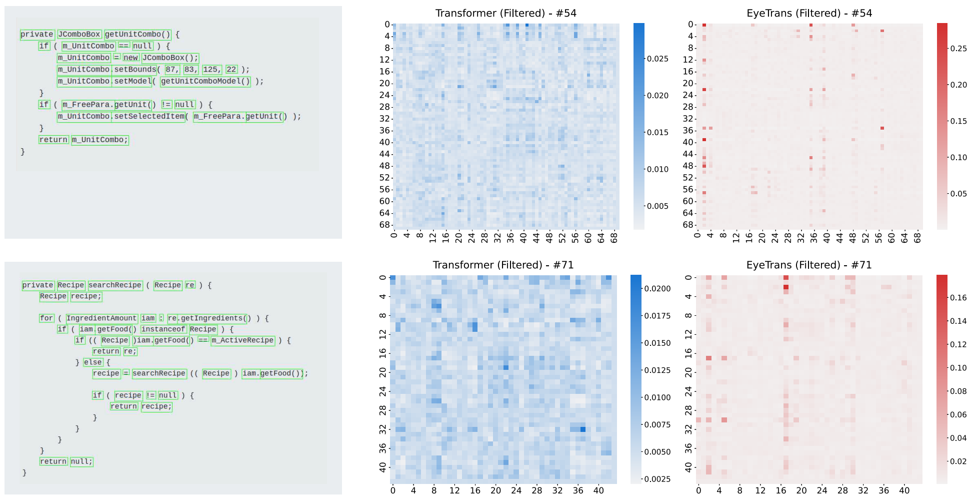
Fig. 7. Learning curve comparison of EyeTrans and Transformer on General Code Summarization with varying eye-tracking data quality. For simplicity, we use 10 epochs for each curve. The average ROUGE of each data point was determined by conducting the experiments five times, using 0, 1, 42, 123, and 12345 as random seeds.

attention data qualities, dropout rates, and Gaussian noise levels, as depicted in Figure 6. The Figure shows dropout rates from 0.1 to 0.5 with intervals of 0.1 from left to right. In each subplot, the x-axis denotes the increase in Gaussian noise.

Greater Robustness We note that robustness against dropout and noise is enhanced after integrating human attention with machine attention in EyeTrans. In Functional Summarization, EyeTrans exhibits improved performance overall when trained under increased difficulty (i.e., higher dropout rate and Gaussian noise), demonstrating greater robustness compared to the vanilla Transformer model, regardless of the human attention data quality.

For answering (2), we plot the ROUGE-1 and ROUGE-2 learning curves for each epoch during the General Code Summarization training process, as shown in Figure 7. For comparison, we use only the first 10 epochs for each curve. In the Figure, we observe a steady improvement from EyeTrans in ROUGE-1 and ROUGE-2, particularly in the later stages of training with the Filtered dataset. For the strictly filtered dataset, due to the decrease in the training set size, both EyeTrans and Transformer struggle in the initial phase of training, making it challenging to distinguish EyeTrans from the baseline Transformer. This indicates it is critical to include both data quality and diversity when filtering human attention data during training.

In summarizing the characteristics of EyeTrans from both RQ2 and RQ3, we conclude: (1) the integration of eye-tracking data into the Transformer enhances the overall performance in code summarization tasks, improving the model's robustness, and (2) there is a pivotal equilibrium between data quality and diversity. The filtered dataset is advantageous for both RQs and generally outperforms the Original dataset, whereas the strictly filtered dataset sacrifices data diversity for quality, leading to reduced performance and training efficiency.



```
private JComboBox getUnitCombo() {
    if ( m_UnitCombo == null ) {
        m_UnitCombo = new JComboBox();
        m_UnitCombo.setBounds( 87, 83, 125, 22 );
        m_UnitCombo.setModel( getUnitComboModel() );
    }
    if ( m_FreePara.getUnit() != null ) {
        m_UnitCombo.setSelectedItem( m_FreePara.getUnit() );
    }
    return m_UnitCombo;
}
```

```
private Recipe searchRecipe ( Recipe re ) {
    Recipe recipe;

    for ( IngredientAmount iam : re.getIngredients() ) {
        if ( iam.getFood() instanceof Recipe ) {
            if (( Recipe )iam.getFood() == m_ActiveRecipe ) {
                return re;
            } else {
                recipe = searchRecipe (( Recipe ) iam.getFood());

                if ( recipe != null ) {
                    return recipe;
                }
            }
        }
    }
    return null;
}
```

Transformer (Filtered) - #54
EyeTrans (Filtered) - #54
Transformer (Filtered) - #71
EyeTrans (Filtered) - #71

(a) Example Code

(b) Without attention switch (Transformer)

(c) With attention switch (EyeTrans)

Fig. 8. Illustration of two examples on Attention Simplification in Functional Summarization. We use heatmaps to visualize the QK^T matrix extracted from the first Transformer block in both models. (a) showcases the example Java code comprehended by the models (green boxes indicate the bounding boxes for eye-tracking data analysis); (b) visualizes the Transformer attention map without attention switch; and (c) visualizes the Transformer attention map with attention switch (EyeTrans). We use models trained with a seed of 42 to ensure the reproducibility of the visualization maps.

5.4 RQ4: Merged Attention Map

In the preceding subsections, we examined the characteristics of eye-tracking data to substantiate our experiments and conducted a quantitative comparison between EyeTrans and Transformer concerning two code summarization tasks. In this subsection, we address the remaining question: specifically, what observable changes occur in the self-attention mechanism of the Transformer structure after incorporating human attention? To answer this question, we use visualizations of the QK^T matrix in the first block of the Transformer as approximations of the model's attention [73]. Specifically, as detailed in Section 2.2, the QK^T matrix in Equation (1) conceptually captures the pair-wise inter-relation between input tokens. Thus, we refer to this matrix as *attention map* and use heatmaps to portray it, effectively representing the Transformer's attention patterns. After examining attention maps from both EyeTrans and a Transformer without human attention information, we identified one interesting pattern which we refer to as *Attention Simplification.*

Attention Simplification We observe that in both Functional Summarization and General Code Summarization, EyeTrans enhances the performance of the Transformer by refining the self-attention pattern. We use the attention map from Functional Summarization as examples for a case study, as shown in Figure 8. In the Figure, EyeTrans suppresses some unimportant relations in the matrix while emphasizing others, intensifying the effectiveness of selected patterns. This illustrates one potential mechanism where human attention provides pivotal guidance, eliminating redundant and useless weights to enhance the performance of the globally perceptive attention maps in the Transformer. Visualizing attention maps for both examples in the case study can reveal changes within the Transformer block caused by the incorporation of human attention, offering insights for future research.

The observed performance enhancement by EyeTrans and the identified refined attention pattern might also provide potential insights for future work, particularly in the domain of explainable AI, shedding light on how human attention can guide model refinement and facilitate the interpretation of model decisions. Furthermore, the implications of these findings might inspire more extensive explorations into the incorporation of human attention in model development, contributing to evolution of more intuitive, understandable, and reliable AI systems.

5.5 Ablation Studies

We conduct further ablation studies under three distinct experimental settings:

(1) Removing the positional/height embedding term from both EyeTrans and Transformer.
(2) Replacing the activation function, ϕ, in Expression (4) from *ReLU* to *Sigmoid*.
(3) Substituting $E_\gamma \odot (\phi(\sum_{\alpha_k=\gamma} P^k_{\alpha_k\beta_k})) + H_\gamma \odot (\phi(\sum_{\beta_k=\gamma} P^k_{\alpha_k\beta_k}))$ in place of Expression (4), removing the +1 in the expression and thus reducing the relative importance of E_γ and H_γ.

We used Functional Summarization with R_2 and N_2 to exemplify the changes. Our observations are as follows: in ablation setting (1), both EyeTrans and the Transformer exhibit a drop by 12.74% and 48.27% in MAF1, respectively, indicating that height embeddings adequately encode positional information into Transformer models to improve performance. In settings (2) and (3), the performance of EyeTrans drops by 15.04% and 32.76% in MAF1, respectively, which provides justification for our model design choices.

6 THREATS TO VALIDITY

There are two main threats to the validity of our evaluation. First, due to the specialized nature of human study research, we created a dataset with eye-tracking information to evaluate EyeTrans, which is based on Java. However, its effectiveness may vary with other programming languages. Second, for both the Functional Summarization and General Code Summarization tasks, the tasks and evaluation metrics used, such as ROUGE and MAF1, may not correlate with human developer performance. To minimize bias as much as possible, we selected multiple metrics and conducted several experiments, using fixed seeds.

7 RELATED WORK

EyeTrans lies at the intersection of code summarization, human attention, and machine learning. In this section, we highlight the relevance of EyeTrans with past works in these domains.

7.1 Code Summarization and Human Attention

Eye-tracking has been used to measure human attention during various Software Engineering tasks [57], including code summarization [3, 51]. This technology is particularly suited to study programmers given its high accuracy [1], and direct integration with developers' working environments [54]. Researchers have used eye-tracking to study debugging behaviors [12], differences between expert and novice coders [3], and code reading strategies [16], among others [43]. Within code summarization research, Rodeghero et al. conducted an experiment similar to the eye-tracking data collection in this study, where programmers' gaze was recorded as they wrote code summaries [51]. That work also aimed to improve methods for automated source code summarization by incorporating human attention. This approach did not use machine learning, instead selecting keywords for summaries based on where programmers fixated most.

In the years since that paper was published, the advancement in deep learning propelled machine learning models to autonomously generate summaries for source code, a task referred to as neural code summarization. Since NeuralCodeSum [5] first introduced the use of Transformers in neural code summarization, many Transformer-based models have showcased remarkable performance across various task settings [20, 21, 58, 65, 69]. Notably, the leading approaches have significantly benefited from the structural information of source code, particularly by leveraging AST representations [7, 20, 21, 26, 36, 38, 59, 65]. EyeTrans elaborates upon this line of neural code summarization work by pioneering the integration of human attention data to improve model performance.

7.2 Integration of Eye-Tracking in Machine Learning

EyeTrans also builds upon past works that leverage eye-tracking to improve general machine-learning performance. In computer vision, several works have demonstrated improved performance by incorporating eye-tracking data [19, 31, 47, 62, 70, 72]. Meanwhile, in NLP, the use of eye-tracking data has been shown beneficial in tasks such as syntactic labeling [34], pronoun classification [71], reference resolution [28], and multi-word expression prediction [52]. These studies exemplify the mainstream methodology in NLP for integrating eye-tracking data, primarily employing it as an additional set of input for the NLP models, distinct and separate from the original dataset.

Another prevalent methodology employs eye-tracking data as either a regularizing factor or an additional task to align neural networks' decision-making with human attention patterns [11, 33, 75]. Recently, a notable contribution from Sood et al. [60] involved modifying the Luong attention layer within an LSTM, by introducing token-specific attention scores that mimic human eye fixations.

EyeTrans advances former works by directly using Transformers' standard input embedding modality to represent human attention, eliminating past works' requirement to either modify the standard NLP model architectures or use eye-tracking data as a separate input set. Moreover, compared to past works' predominant focus on LSTM-centric networks, we pioneer the integration of human attention into modern Transformer-based architectures. We also pioneer the use of attention switches, rather than solely relying on fixation, as a representation of human attention to enhance the performance of machine learning models.

8 FUTURE WORKS

We consider EyeTrans as a first step towards integrating human and machine attention. As such, many potential extensions and ramifications of this work are yet to be explored. We discuss such future directions in this section.

We intend EyeTrans to work in concert with Large Language Models (LLMs), not to compete with them. The key concepts of our paper is applicable to many Transformer-based models. The majority of LLMs today are fundamentally based on the Transformer backbone, but with more attention heads, a wider embedding vector size, scale increases, and other changes. We focus on improving the underlying Transformer architecture in this paper. The benefits we propose could in theory be applied to larger models, though we test them using small eye-tracking data and smaller models. This aligns with the current state of the art of eye-tracking studies in SE, which is often coupled with limitations on sample sizes.

Yet, there may exist realistic approaches to overcome the costly acquisition of human visual attention data. For example, in reading natural languages (as compared to code), the E-Z Reader model [48] has been well-established in predicting human visual gaze. Applying E-Z-Reader-predicted pseudo-human-gaze on natural text, and consequently aligning NLP attention according to such pseudo-human-gaze, has improved model performance on natural languages [60]. The SE community currently lacks well-established computational modeling of programmers' gaze during code reading. However, steps towards this direction have recently been undertaken [9]. With future computational modeling techniques capable of adding accurate pseudo-visual attention to code, EyeTrans can be trained on regular, large datasets just like any other NLP model, without the need for extensive eye-tracking experiments.

9 CONCLUSION

In this paper, we present EyeTrans, an approach to effectively incorporate human attention into the Transformer for neural code summarization tasks. In EyeTrans, human attention serves as a "connection" between two ends of the token embeddings, representing a data-centric incorporation of human attention without altering the Transformer structure. Integrating human attention in training results in a performance improvement of up to 29.91% in Functional Summarization, and up to 6.39% in General Code Summarization, thereby demonstrating the effectiveness of EyeTrans. This is the first, proof-of-concept work to integrate the eye-tracking patterns of programmers into Transformer models, achieving improvement on performance across two different code summarization tasks and comprehensive analysis.

We hope the basic concept of EyeTrans can be applied to numerous training scenarios that use the Transformer or Transformer block as a fundamental component. Moreover, with the future development of pseudo-eye-tracking paths and data augmentation on other structured and human-readable data in programming, our idea can be readily adopted. This will allow for the incorporation of a new modality during the training of the Transformer and has the potential to become a fundamental component of the Transformer. In the future, we plan to enhance EyeTrans by designing paired eye-tracking data augmentation methods, such as few-shot link prediction and label propagation, into the general model structure.

10 DECLARATIONS

In this section, we outline the compliance, data availability, funding sources, and acknowledgments associated with this study, highlighting the essential aspects that underpin our research.

Ethical Compliance This study was conducted in compliance with all applicable ethical standards and received approval from the Institutional Review Board (IRB). Informed consent was obtained from all individual participants involved in the study.

Data-Availability Statement All data and scripts are available on Zenodo [74]. The repository includes comprehensive documentation to facilitate replication and extension of our research.

Funding Statement This research was supported by NSF CCF-2211429, NSF CCF-2211428, NSF CCF-2100035 and NSF SaTC-2312057.

Acknowledgments We appreciate the anonymous reviewers' feedback and constructive suggestions for improving our manuscript. We also greatly thank all the participants in our study for their participation.

REFERENCES

[1] 2023. https://go.tobii.com/tobii-pro-fusion-user-manual
[2] Nahla J Abid, Jonathan I Maletic, and Bonita Sharif. 2019. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*. 1–9.
[3] Nahla J Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I Maletic. 2019. Developer reading behavior while summarizing java methods: Size and context matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 384–395.
[4] Hammad Ahmad, Zachary Karas, Kimberly Diaz, Amir Kamil, Jean-Baptiste Jeannin, and Westley Weimer. 2023. How Do We Read Formal Claims? Eye-Tracking and the Cognition of Proofs about Algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 208–220.
[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).
[6] Nasir Ali, Zohreh Sharafi, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2015. An empirical study on the importance of source code entities for requirements traceability. *Empirical software engineering* 20 (2015), 442–478.
[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[8] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[9] Aakash Bansal, Bonita Sharif, and Collin McMillan. 2023. Towards Modeling Human Attention from Eye Movements for Neural Source Code Summarization. *Proceedings of the ACM on Human-Computer Interaction* 7, ETRA (2023), 1–19.

[10] Aakash Bansal, Chia-Yi Su, Zachary Karas, Yifan Zhang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2023. Modeling Programmer Attention as Scanpath Prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1732–1736.

[11] Maria Barrett, Joachim Bingel, Nora Hollenstein, Marek Rei, and Anders Søgaard. 2018. Sequence classification with human attention. In *Proceedings of the 22nd conference on computational natural language learning*. 302–312.

[12] Roman Bednarik. 2012. Expertise-dependent Visual Attention Strategies Develop over Time During Debugging with Multiple Code Representations. *International Journal of Human-Computer Studies* 70, 2 (Feb. 2012), 143–155. https://doi.org/10.1016/j.ijhcs.2011.09.003

[13] Jean-Francois Bergeretti and Bernard A Carré. 1985. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 37–61.

[14] Birtukan Birawo and Pawel Kasprowski. 2022. Review and evaluation of eye movement event detection algorithms. *Sensors* 22, 22 (2022), 8810.

[15] Egon Börger and Wolfram Schulte. 1999. A programmer friendly modular definition of the semantics of Java. *Formal Syntax and Semantics of Java* (1999), 353–404.

[16] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265.

[17] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*. IEEE, 516–519.

[18] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The Roles Beacons Play in Comprehension for Novice and Expert Programmers.. In *PPIG*. 5.

[19] Zhengcong Fei. 2022. Attention-aligned transformer for image captioning. In *proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 607–615.

[20] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code Structure–Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[21] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–24.

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

[23] Meng-Hao Guo, Tian-Xing Xu, Jiang-Jiang Liu, Zheng-Ning Liu, Peng-Tao Jiang, Tai-Jiang Mu, Song-Hai Zhang, Ralph R Martin, Ming-Ming Cheng, and Shi-Min Hu. 2022. Attention mechanisms in computer vision: A survey. *Computational visual media* 8, 3 (2022), 331–368.

[24] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 223–226.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[27] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. 2020. Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 456–468.

[28] Ryu Iida, Masaaki Yasuhara, and Takenobu Tokunaga. 2011. Multi-modal reference resolution in situated dialogue by integrating linguistic and extra-linguistic clues. In *Proceedings of 5th International Joint Conference on Natural Language Processing*. 84–92.

[29] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2073–2083.

[30] Marcel A Just and Patricia A Carpenter. 1980. A theory of reading: from eye fixations to comprehension. *Psychological review* 87, 4 (1980), 329.

[31] Nour Karessli, Zeynep Akata, Bernt Schiele, and Andreas Bulling. 2017. Gaze embeddings for zero-shot image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4525–4534.

[32] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[33] Sigrid Klerke, Yoav Goldberg, and Anders Søgaard. 2016. Improving sentence compression by learning to predict gaze. *arXiv preprint arXiv:1604.03357* (2016).

[34] Sigrid Klerke and Barbara Plank. 2019. At a glance: The impact of gaze aggregation views on syntactic tagging. In *Proceedings of the Beyond Vision and LANguage: inTEgrating Real-world kNowledge (LANTERN)*. 51–61.

[35] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. 2022. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 520–532.

[36] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[37] Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660* (2019).

[38] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 184–195.

[39] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[41] Sun Developer Network. 1999. Code conventions for the Java programming language.

[42] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. 2021. A review on the attention mechanism of deep learning. *Neurocomputing* 452 (2021), 48–62.

[43] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. 2018. A Survey on the Usage of Eye-Tracking in Computer Programming. *ACM Comput. Surv.* 51, 1, Article 5 (Jan. 2018), 58 pages. https://doi.org/10.1145/3145904

[44] Anneli Olsen. 2012. The Tobii I-VT fixation filter. *Tobii Technology* 21 (2012), 4–19.

[45] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? an eye tracking study. In *Proceedings of the 28th International Conference on Program Comprehension*. 342–353.

[46] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277* (2023).

[47] Tingting Qiao, Jianfeng Dong, and Duanqing Xu. 2018. Exploring human-like attention supervision in visual question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[48] Erik D Reichle, Keith Rayner, and Alexander Pollatsek. 2003. The EZ Reader model of eye-movement control in reading: Comparisons to other models. *Behavioral and brain sciences* 26, 4 (2003), 445–476.

[49] Paige Rodeghero, Cheng Liu, Paul W McBurney, and Collin McMillan. 2015. An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054.

[50] Paige Rodeghero and Collin McMillan. 2015. An empirical study on the patterns of eye movement during summarization tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[51] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

[52] Omid Rohanian, Shiva Taslimipoor, Victoria Yaneva, and Le An Ha. 2017. Using gaze data to predict multiword expressions. (2017).

[53] Herbert Schildt. 2007. Java: the complete reference. (2007).

[54] Timothy R Shaffer, Jenna L Wise, Braden M Walters, Sebastian C Müller, Michael Falcone, and Bonita Sharif. 2015. itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 954–957.

[55] Zohreh Sharafi, Yu Huang, Kevin Leach, and Westley Weimer. 2021. Toward an Objective Measure of Developers' Cognitive Activities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–40.

[56] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. 2015. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 96–103.

[57] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering* 25 (2020), 3128–3174.

[58] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[59] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987* (2021).

[60] Ekta Sood, Simon Tannert, Philipp Müller, and Andreas Bulling. 2020. Improving natural language processing tasks with human gaze-guided neural attention. *Advances in Neural Information Processing Systems* 33 (2020), 6327–6341.

[61] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[62] Yusuke Sugano and Andreas Bulling. 2016. Seeing with humans: Gaze-assisted neural image captioning. *arXiv preprint arXiv:1608.05203* (2016).

[63] Jerry Chih-Yuan Sun and Kelly Yi-Chuan Hsu. 2019. A smart eye-tracking feedback scaffolding approach to improving students' learning self-efficacy and performance in a C programming course. *Computers in Human Behavior* 95 (2019), 66–72.

[64] Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. 2021. AST-transformer: Encoding abstract syntax trees efficiently for code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1193–1195.

[65] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. 2022. AST-trans: Code summarization with efficient tree-structured attention. In *Proceedings of the 44th International Conference on Software Engineering*. 150–162.

[66] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

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

[68] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13.

[69] Hongqiu Wu, Hai Zhao, and Min Zhang. 2020. Code summarization with structure-induced transformer. *arXiv preprint arXiv:2012.14710* (2020).

[70] Jia Xu, Lopamudra Mukherjee, Yin Li, Jamieson Warner, James M Rehg, and Vikas Singh. 2015. Gaze-enabled egocentric video summarization via constrained submodular maximization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2235–2244.

[71] Victoria Yaneva, Le An Ha, Richard Evans, and Ruslan Mitkov. 2020. Classifying referential and non-referential it using gaze. *arXiv preprint arXiv:2006.13327* (2020).

[72] Youngjae Yu, Jongwook Choi, Yeonhwa Kim, Kyung Yoo, Sang-Hun Lee, and Gunhee Kim. 2017. Supervising neural attention models for video captioning by human gaze data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 490–498.

[73] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[74] Yifan Zhang. [n. d.]. Reproduction Package for the FSE 2024 Paper "EyeTrans: Merging Human and Machine Attention for Neural Code Summarization". https://doi.org/10.5281/zenodo.10684985

[75] Yingyi Zhang and Chengzhi Zhang. 2019. Using human attention to extract keyphrase from microblog post. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 5867–5872.

[76] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.

[77] Wenyu Zhu, Hao Wang, Yuchen Zhou, Jiaming Wang, Zihan Sha, Zeyu Gao, and Chao Zhang. 2023. kTrans: Knowledge-Aware Transformer for Binary Code Embedding. *arXiv preprint arXiv:2308.12659* (2023).

Received 2023-09-28; accepted 2024-01-23