CombTransformers: Statement-Wise Transformers for Statement-Wise Representations

Francesco Bertolotti and Walter Cazzola

***Abstract*—This study presents a novel category of Transformer architectures known as comb transformers, which effectively reduce the space complexity of the self-attention layer from a quadratic to a subquadratic level. This is achieved by processing sequence segments independently and incorporating $\mathcal{X}$-word embeddings to merge cross-segment information. The reduction in attention memory requirements enables the deployment of deeper architectures, potentially leading to more competitive outcomes. Furthermore, we design an abstract syntax tree (AST)-based code representation to effectively exploit comb transformer properties. To explore the potential of our approach, we develop nine specific instances based on three popular architectural concepts: funnel, hourglass, and encoder-decoder. These architectures are subsequently trained on three code-related tasks: method name generation, code search, and code summarization. These tasks encompass a range of capabilities: short/long sequence generation and classification. In addition to the proposed comb transformers, we also evaluate several baseline architectures for comparative analysis. Our findings demonstrate that the comb transformers match the performance of the baselines and frequently perform better.**

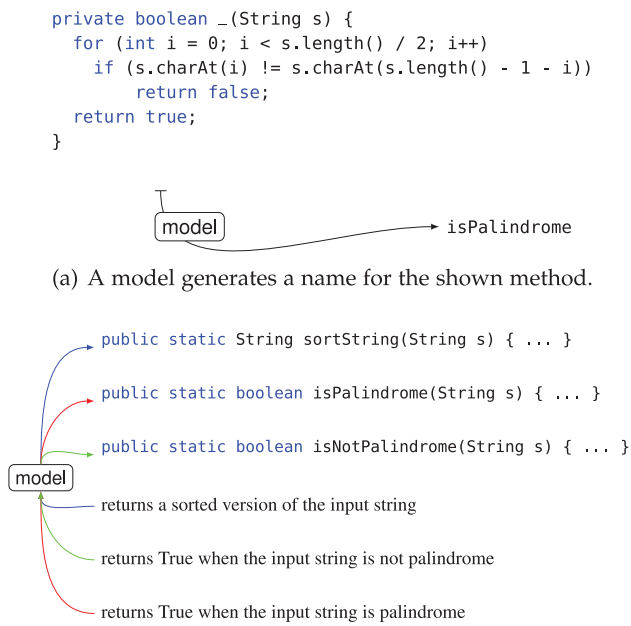
***Index Terms*—Programming languages, machine learning, learning representations, code search and summarization, method name Gen.**

I. Introduction

OVERVIEW. Effective code representations and effective neural architectures for programming language processing are becoming more and more relevant for developing tools that leverage the potential of deep learning on code. Such representations and architectures have the potential to improve the current state-of-the-art on a variety of tasks, which include: *automatic code generation* [1], [2], [3], *automatic code completion* [4], [5], [6], *extreme code summarization* [7], [8], [9], *automatic test generation* [10], [11], *automatic bug detection* [12], [13], [14], *code ranking* [15], [16] and many others. For some of these tasks, we can leverage the availability of massive datasets to train neural networks (NN).

In this work, we focus on three tasks: method name generation, code ranking, and code summarization.

Method Name Generation. Given a method body, the model is required to generate a string representing its name (see Fig. 1(a)). Good method names provide a natural language summary of the method body. Such a model can be used to automatically inspect codebases without human intervention. Moreover, the design of good method names is shown to be a critical issue [17], [18], [19] by itself. Fig. 1(a) shows a method. Given these snippets as input, the model is expected to output a relevant method name. In this case, the model is required to output the name *isPalindrome*. Codetovec [9] and Codetoseq [8] introduced two architectures for this task. Both architectures exploit a specific representation of source code—leaf-to-leaf abstract syntax tree paths. Differently, Fold2vec [20] focuses on



(a) A model generates a name for the shown method.

(b) Three methods correctly associated with their descriptions

(c) A method with documentation (comment) model generated.

Fig. 1. Three code-related task.

Manuscript received 19 September 2022; revised 11 August 2023; accepted 28 August 2023. Date of publication 6 September 2023; date of current version 17 October 2023. This work was supported in part by the MUR project "T-LADIES" under Grant PRIN 2020TL3X8X. Recommended for acceptance by T. Menzies. *(Corresponding author: Walter Cazzola.)*

The authors are with the Department of Computer Science, Università degli Studi di Milano, 20133 Milan, Italy (e-mail: francesco.bertolotti@unimi.it; cazzola@di.unimi.it).

Digital Object Identifier 10.1109/TSE.2023.3310793

encoding statements independently from each other to obtain a compressed representation of the method.

Code Ranking. Given a natural language query and a set of methods, the model is required to match the query with the most relevant method. A significant portion of the software development is devoted to adapt already existing code to new use cases. Therefore, to effectively suggest source code snippets, given a natural language query, has the potential to boost productivity. Fig. 1(b) shows three methods correctly associated with the respective queries. Gu et al. [21] represent methods as triplets of: tokenized method name, tokenized API calls, and set of body tokens. This representation is used to train a *recurrent neural architecture*. Husain et al. [15] represent the method as a sequence of tokens. It has been used to train several types of architecture.

Code Summarization. The goal of the task is to generate a concise natural language description for a given code snippet (Fig. 1(c)). Understanding pre-existing code snippets is a challenging task even for experienced programmers, and the process of manually writing documentation is often considered tedious. Consequently, the automation of this task is highly desirable. Neural networks have been employed to tackle the code summarization task. One notable example is CodeBERT [22] a Transformer-based architecture pretrained with masked language modeling [23]. During pretraining CodeBERT learns to generate documentation. A more advanced approach is applied in [24], [25] which introduces more self-supervised objectives during pretraining.

In this work, to make a fair comparison between architectures, we focus on end-to-end training: every architecture is trained on the same data, with the same objectives for the same amount of time. We believe that this setting allows for a fair comparison between neural architectures, which is our goal. On the other hand, the reader should note that pretraining with self-supervised tasks and fine-tuning for downstream tasks is one of the most effective ways to achieve state-of-the-art performance in neural networks.

Instead of designing a code representation that describes the method globally, we argue that a code representation that describes methods as sequence of statements can have few advantages:

1. it can be used to reduce the space complexity mechanism of the self-attention layers;
2. allowing to process longer methods, with;
3. deeper architectures.

The main contributions of this work are:

- a class of Transformers architecture, dubbed *CombTransformers*. These architectures have subquadratic memory requirements whereas traditional Transformer architecture has quadratic scaling laws; and
- a novel statement-based code representation, that combined with the CombTransformer architectures, achieves state-of-the-art results on the evaluated tasks.

The rest of the paper is organized as follows. Section II presents recurrent concepts used in this work. Section III gives a detailed overview of the employed statement-based representation. Section IV describes the CombTransformer neural architecture. Section V discusses the results achieved by combining the new representation with the new neural architectures. Finally, Sections VIII and IX give an overview of the related works and present our conclusions respectively.

II. Background

In this section, we will go through some of the terminology and concepts that are recurrent in this work.

Abstract Syntax Tree (AST). Following the definition used by Alon et al. [8], an AST is a tree-shaped representation of a program. It can be formally defined as a quintuple $(N, T, X, s, \delta, \psi)$ where:

- N is the set of non-terminal nodes of the AST. Additionally split N into N_S and N_E. Where, N_S contains statement like nodes. While, N_E contains expression like nodes.
- T is the set of terminal nodes of the AST.
- X is the set of values that terminal nodes can assume.
- $s \in N$ is the root of the AST.
- $\delta : N \to (N \cup T)^+$ is the parent function that maps each non-terminal to its children.
- $\psi : T \to X$ is the value function that maps each terminal to its value.

Feed Forward Network. A Feed Forward Network (FFN) is one of the most common layer in any neural architecture. With this term, we intend a linear layer followed by an activation unit: $y = g(xA^T + b)$ Where, $x \in \mathbb{R}^{1\times d}$ is a input feature vector. $A \in \mathbb{R}^{d'\times d}$ is a matrix of learnable parameters (weights). $b \in \mathbb{R}^{1\times d'}$ is another vector of learnable parameters (bias). Lastly, g is an activation function.

Embedding Layer. An embedding layer is a map $E : I_N \to \mathbb{R}^d$. Where $I_N = \{i \in \mathbb{N} : i < N\}$. We can simply represent the embedding layer with the matrix $E = \mathbb{R}^{N\times d}$. The embedding layer is often used to map tokens (such as «image» or «dog») to a vector of trainable parameters. During training, the NN learns to represent tokens in relation to the others [26].

Global Attention. Global Attention [27] (GA) is an attention mechanism that combines feature vectors through a weighted sum. Let $x_1, x_2, \ldots, x_n \in \mathbb{R}^{d\times 1}$ be the input vectors. We define a vector of trainable parameters $a \in \mathbb{R}^{d\times 1}$. We obtain the combined feature vector $\hat{x} \in \mathbb{R}^{d\times 1}$ as:

$$\hat{x} = \sum_{i=1}^{n} \alpha_i x_i$$

$$\alpha_i = \texttt{softmax}(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}} \qquad s_i = a \cdot x_i^T$$

where, s_i can be see an attention score of x_i, that is, how much attention the NN wants to pay to x_i. α_i are the normalized attention scores, so that, $\sum_i \alpha_i = 1$. Finally, $\hat{x}$ is the resulting vector from the GA. We often say that the vector a *attends* to vectors $x_0, \ldots, x_n$.

Self Attention. Self Attention [28] (SA) is another attention mechanism. Let $x_1, x_2, \ldots, x_n \in \mathbb{R}^{d\times 1}$ be the input vectors. Here, each input vector x_i plays the role of the vector a in GA. Intuitively, each x_i is scored (through dot product) against

the whole input sequence (i.e. each vector x_i attends to each vector x_j). Scores are used, once again, in a weighted sum to produce one of the resulting vectors z_i. Formally, SA plays out as follows.

$$Q = X \cdot W_Q \quad K = X \cdot W_K \quad V = X \cdot W_V.$$

$$Z = Attention(Q, K, V)$$

$$Attention\ (Q, K, V) = \texttt{softmax}\left(\frac{Q \cdot K^T}{d}\right) \cdot V$$

Where, X is the matrix obtained by concatenating the input sequence. Z is the matrix obtained by concatenating the output sequence. W_Q, W_K, W_V are matrices of trainable parameters. The vector sequences denoted by Q, K, and V are also known as query vectors, keys vectors, and value vectors. For the sake of simplicity, we have intentionally left out some details. Please refer to the original paper [28] for more details. Notice that $Q \cdot K^T$ computes the attention scores by multiplying each query vector against each key vector. Instead, the softmax function normalizes the scores. The last matrix multiplication implicitly applies the weighted sum of the normalized scores to the value matrix V. The score matrix (given by $Q \cdot K^T$) has size $\mathcal{O}(n^2)$.

Multi Head Self Attention (MHSA). MHSA [28] is an extension of SA. Intuitively MHSA repeat the SA process multiple times. Let h be an hyper-parameter of choice controlling the number of heads. Let $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times (d/h)}, W^O \in \mathbb{R}^{d \times d}$ with $1 \leq i \leq h$ be matrices of trainable parameters. MHSA plays out as follows:

$$MultiHead(Q, K, V) = Concat(hd_1, hd_2, \ldots, hd_h) \cdot W^O$$

$$hd_i = Attention(Q \cdot W_i^Q, K \cdot W_i^K, V \cdot W_i^V)$$

Intuitively, MHSA repeats the SA layer h times with embeddings of size d/h instead of d. Each query/key/value is projected by the W_i linear transformation in lower dimensional space of size d/h. Next, the attention function is applied across all heads, obtaining hd_i. Finally, we concatenate all the heads (hd_i). The application of the W_O linear transformation yields the output of MHSA mechanism.

Transformer. The Transformer neural architecture [28] is a successful and popular auto-regressive NN that is mostly used in the field of natural language processing (NLP) [22], [23], [29]. The Transformer heavily relies on repeating the layers of MHSA. Despite being a powerful architecture, one of its principal drawbacks is the intense need of memory $\mathcal{O}(n^2)$ for the SA layers, where n is the sequence length. Such a limitation bounds the Transformer to process only short sequences. Finally, we consider as the Transformer the model proposed by Vaswani et al. [28].

Efficient Transformer. The space complexity of the SA mechanism ($\mathcal{O}(n^2)$) and the success of the Transformer architecture has led many efforts to design less costly architectures. Some important advancements are:

– Strided attention [30]. Here, each token can attend only to its close neighborhood.

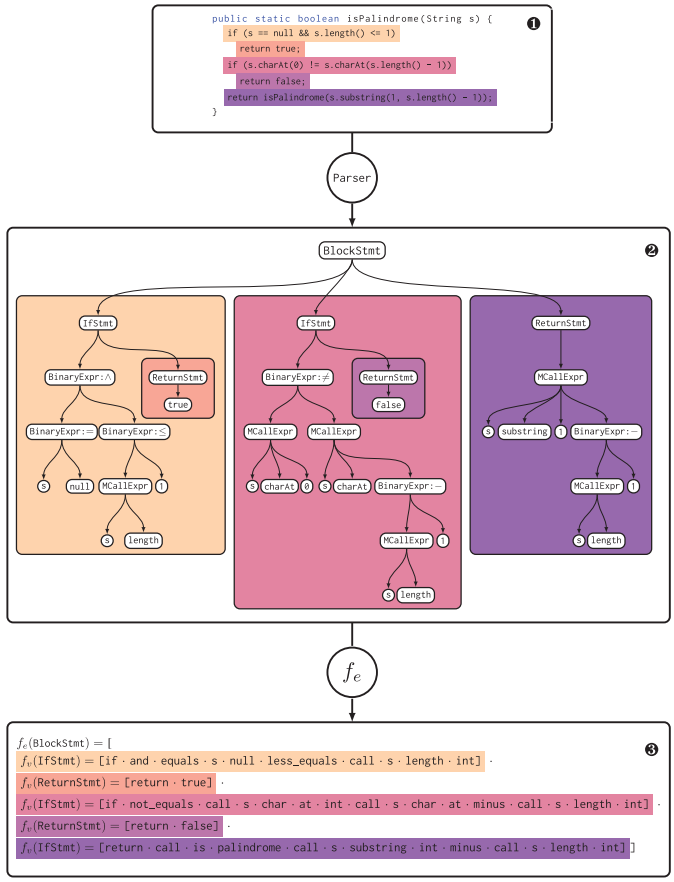


Fig. 2. A Java method (Panel ❶), its AST (Panel ❷), its three CUs (Panel ❸). Colors highlight the process through which statements go.

– Strided-Dilated attention [31]. Again, each token can attend only to its close neighborhood. However, some close tokens are skipped.
– Globally-attending tokens [31]. Here, few tokens are allowed to attend to all other tokens.

III. REPRESENTATION

This section introduces the process to obtain the statement-wise representation from AST. Relying on the AST makes the process independent from the underlying programming language as long as a parser for the language is available.

Tree Unfolding (TU). A TU is the serialization of a given tree obtained by applying a serialization function

$$f : (N \cup T)^+ \rightarrow (N \cup X)^*$$

to the tree. For example, possible TUs are shown in Fig. 3 where f denotes the serialization operation.

Contextual Unfolding (CU). A CU is a specific kind of TU designed to deal with the AST representation of a code snippet. Fig. 2 shows an example. Let us consider the code snippet in panel ①. The AST shown in panel ② is obtained by parsing. The function f_e is applied to the root node of the AST, i.e., node `BlockStmt`. f_e visits the whole AST in a pre-order manner. Every time f_e discovers the root node of a statement tree (e.g.,

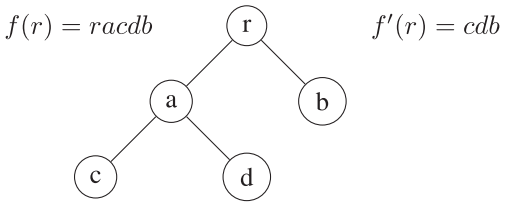


Fig. 3. Two TUs: f applies a pre-order visit, f' returns only leaves.

`ReturnStmt` and `IfStmt`) it applies f_v to the discovered node. The formal definition for f_e is:

$$f_e : (N \cup T)^+ \rightarrow (N^* \times X^*)^+$$

$$f_e(x) = \begin{cases} \varepsilon & \text{if } x \in T \\ \texttt{concat}\{f_e(c) | \forall c \in \delta(x)\} & \text{if } x \in N_E \\ [f_v(x)] \cdot \texttt{concat}\{f_e(c) | \forall c \in \delta(x)\} & \text{otherwise} \end{cases}$$

Similarly to f_e, f_v does a pre-order visit. However, f_v does not visit any tree associated to another statement. Each node visited by f_v becomes a token in the CU. There are as many CUs as statements in the code snippet and each CU is produced by applying f_v. Formally f_v is defined as:

$$f_v(x) = \begin{cases} \psi(x) & \text{if } x \in T \\ x \cdot \texttt{concat}\{f_v(c) | \forall c \in \delta(x) \setminus N_S\} & \text{otherwise.} \end{cases}$$

Finally, it is worth mentioning that some additional processing is done to terminal values. Numbers are removed (e.g., in Fig. 2 '`1`' does not appear in the second CU despite it appears in the AST). Every terminal value is split into sub-tokens (e.g., the token '`isPalindrome`' is split into two tokens: '`is`' and '`palindrome`'). These choices permit to limit the vocabulary's size. We employed spiral [32] as token splitter, but possibilities are available [33], [34]. Instead, as a parser, we have employed `JavaParser` [35]. Lastly, an additional CU representing the method declaration stripped off the method name is stacked on top of the others.

IV. Neural Architectures

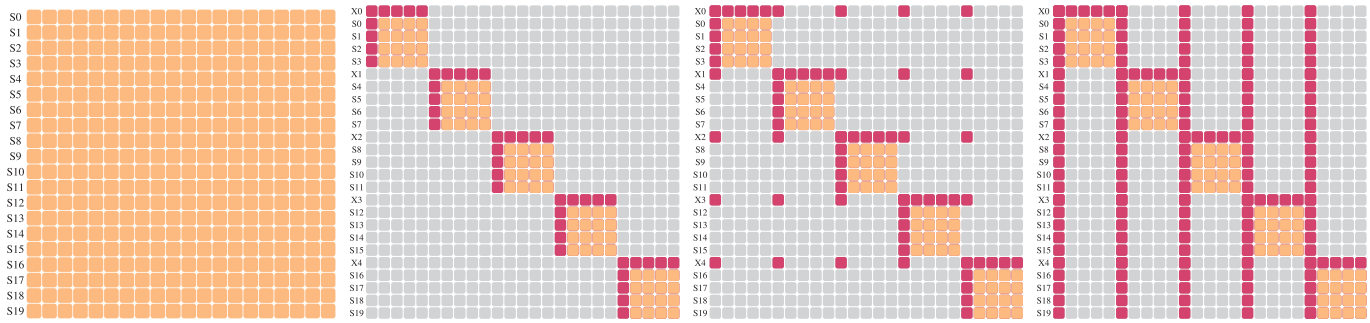
One of the most impactful architectures is the Transformer [28]. One of the most discussed points of the Transformer is the SA layer as it heavily impacts memory and computation wrt. the sequence length. The simplest way to lighten the memory impact is to reduce the sequence length which may remove useful information. A workaround is to split the sequence into segments and process the segments separately. To merge segment information, either the $\mathcal{X}$-word embedding mechanism or specific attention patterns as those in Fig. 4 can be used. Depending on the choice of segment length s and the number of segments k, one can prove sub-quadratic memory scaling laws (notice that, if n is the sequence length, it must hold that $n = sk$). In general, a small s stresses the merging mechanism. Instead, a large s stresses the required memory. A reasonable trade-off is using $k = \sqrt{n}$ segments each of $s = \sqrt{n}$ tokens. Thus requiring $\mathcal{O}(ks^2) = \mathcal{O}(n\sqrt{n})$ memory. We present nine architectures that combine these ideas in different ways.

Notation. Let us define some common notation.

- S denotes the full input sequence. $S \in \mathbb{R}^{n \times d}$. Where, n is the sequence length, and d is the embedding dimension. S is obtained from an embedding layer applied to a token sequence as in [23].
- To split matrices, we will use a Python inspired slicing notation. E.g., $S[:c, -b:]$ denotes the first c vector embeddings in the matrix S with only the last b elements.
- S_1,...$S_k \in \mathbb{R}^{s \times d}$ denote the k segments of S, each of size s (when $n = sk$). Thus $S_i = S[is:(i+1)s]$.
- $X \in \mathbb{R}^{t \times d}$ denotes the set of $t\mathcal{X}$-word embeddings. $\mathcal{X}$-words are obtained from an embedding layer applied to special tokens as in [36]. The value of t is assumed to be constant and $t << n$.
- VE_p^l denotes the application of a Transformer encoder layer l times with attention pattern p (see Fig. 4). E.g., VE_2^3 represents the application of a Transformer encoder layer 3 times with attention pattern from Fig. 4(c). When $p = 0$, the attention pattern is equivalent to the one used in the original Transformer architecture [28].
- VD^l denotes the application of a vanilla decoder layer l times as in [28]. Each application of VD(A,B) internally applies a MHSA(A,A) and a MHSA(A,B) layer.
- $\mathcal{X}$-word embeddings are extra embeddings added to each segment, often denoted as $\mathcal{X}$s.

Attention Patterns. Fig. 4 depicts four potential attention patterns. Fig. 4(a) illustrates the traditional Self-Attention (SA) layer, where each embedding attends to all other embeddings, resulting in a quadratic complexity due to its pairwise nature. However, it is often observed that embeddings primarily focus on their immediate neighbors for attention [31], suggesting an opportunity for memory optimization. Additionally, exploiting the inherent structure of source code being organized into statements allows us to naturally divide the sequence into segments. Consequently, an embedding can attend to another embedding only if they belong to the same statement in the patterns shown as Fig. 4(b), 4(c), and 4(d) (orange tiles). Cross-segment attention, represented by gray tiles, is prohibited, leading to memory efficiency. Notably, in pattern Fig. 4(b), inter-segment information flow is constrained. Conversely, pattern Fig. 4(c) permits information exchange through $\mathcal{X}$-word embeddings between segments. Here, $\mathcal{X}$s within a segment can attend to all $\mathcal{X}$s across the sequence, and they can also attend to embeddings within their own segment (depicted in purple). Extending from the previous pattern, pattern Fig. 4(d) allows embeddings to attend to all $\mathcal{X}$s as well, facilitating accelerated cross-segment information propagation. The memory requirements for patterns Fig. 4(b), 4(c), and 4(d) are determined by the segment size (s) and the number of segments (k), resulting in a complexity of $\mathcal{O}(ks^2)$.

Funnel Architecture. Funnel architectures and layers are used to compress large dimensional inputs into a lower dimensional space. The main intuition behind the funnel architecture is that often inputs have redundant information. By eliminating the redundancy, we can boost the performance of



(a) attention pattern 0 (b) attention pattern 1 (c) attention pattern 2 (d) attention pattern 3

Fig. 4. Three possible attention patterns involving $\mathcal{X}$-word embeddings compared to the full attention mechanism (pattern 0). A colored square in position (i, j) means that token i is allowed to attend to token j. Vice versa, a gray token in position (i, j) means that the token i cannot attend to token j. Magenta squares are attention involving a $\mathcal{X}$s. Instead, orange squares are attentions involving only tokens in the original sequence.

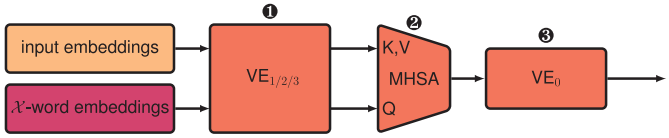


Fig. 5. Input embedding and $\mathcal{X}$-word embedding feed an encoder architecture. The encoder applies several attention layer. Attention layer uses one of the attention pattern in Fig. 4. This layer is followed by an MHSA layer using $\mathcal{X}$-word embeddings to reduce the sequence length. Finally, a Transformer encoder layer is applied to the reduced sequence.

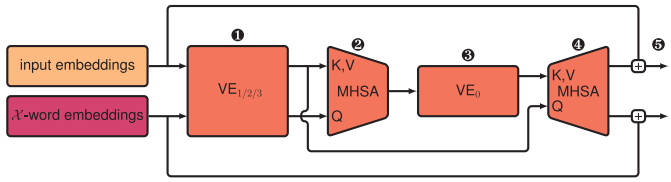


Fig. 6. Input embedding and $\mathcal{X}$-word embedding feed an encoder layer. The encoder applies several attention layer with one of the presented patterns (see Fig. 4). A MHSA layer is used to reduce the sequence length. A Transformer encoder is applied to the reduced sequence. An upscaled version of the reduced sequence is obtained by applying another MHSA. Finally, we introduce a skip connection.

the architecture without compromising the quality of the results [37]. Fig. 5 depicts the architecture.

To use token segmenting and $\mathcal{X}$-word embeddings to implement a funnel architecture, firstly, we have to concatenate the $\mathcal{X}$-word embeddings with the segments:

$$M_1 = concat(X, S_1), \dots, M_k = concat(X, S_k)$$

Now, each $M_i \in \mathbb{R}^{(s+t)\times d}$. Next, we apply an encoder layer:

$$M_1^l = \text{VE}_0^l(M_1), \dots, M_k^l = \text{VE}_0^l(M_k)$$

Each application of $\text{VE}_0^1(M_i)$ has a space complexity for the SA mechanism of $\mathcal{O}\big((s+t)^2\big)$. Since $\text{VE}_0^l(M_i)$ is applied k times per layer, we have a space complexity of $\mathcal{O}\big(k(s+t)^2\big)$.

In practice, the application of VE_0^1 on all segments produces an attention pattern equivalent to the one shown in Fig. 4(b). However, also patterns Fig. 4(c) and 4(d) have the same memory requirement of $\mathcal{O}\big(k(s+t)^2\big)$ (as in Fig. 5-❶). Next, we apply a MHSA layer between $\mathcal{X}$-words and sequence tokens (the funnelling layer, Fig. 5-❷):

$$\begin{aligned} G_0 &= \text{MHSA}(M_0^l[:t], M_0^l[t:]) \\ &\vdots \\ G_k &= \text{MHSA}(M_k^l[:t], M_k^l[t:]) \end{aligned}$$

Again, since $M_i^l[:t]$ has length t and $M_i^l[t:]$ has length s, this operation has an attention space complexity of $\mathcal{O}(kts)$. Now, each G_i block is composed of t embeddings, each of size d. By concatenating all G_i blocks:

$$G = concat(G_0, \dots, G_k)$$

As a result, G is composed of kt embeddings. Finally, we apply additional VE_0^l layer to G (Fig. 5-❸), obtaining: $G^l = \text{VE}_0^l(G)$. Since G has length of kt, the space complexity of the SA is $\mathcal{O}\big((kt)^2\big)$. Here, G^l is a compressed version of the original sequence. G^l can be fed to a decoder architecture as in [28], or directly used to make predictions as in [15].

Hourglass Architecture. Hourglass architecture and layers are used in variety of applications [38], [39]. They compress the input dimension into a lower one. Next, they upscale the lower dimension into the original input dimension. This structure allows for a residual connection [40] which usually leads to better generalizations [41]. In this section, we will discuss the implementation of an hourglass layer using the concepts of segmenting and $\mathcal{X}$-word embeddings. A depiction of the architecture is shown in Fig. 6.

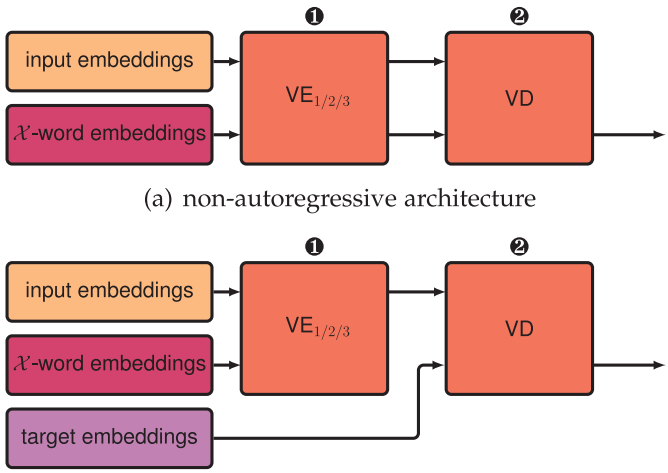
Our hourglass architecture builds upon the previously defined funnel architecture. Let us start from the previously defined G^l (Fig. 5-❷). To restore the input dimension, we apply a MHSA layer between S and G (Fig. 6-❹):

$$S' = \text{MHSA}(S, G)$$

Recall that the length of G^l was kt, while the length of S was n. Then, the space complexity of the attention mechanism of this layer is $\mathcal{O}(nkt)$. As S' has the same dimension of S, we can add a residual connection between S and S' (Fig. 6-❺):

$$S'' = S + S'$$

Finally, we can stack this layer to get a deeper architecture.



(a) non-autoregressive architecture

(b) autoregressive architecture

Fig. 7. Input and $\mathcal{X}$-word embeddings feed an encoder. The encoder applies several attention layer that use the attention patterns in Fig. 4. The resulting sequence is fed to Transformer decoder layer.

Encoder-Decoder Architecture. We propose architectures for autoregressive (Fig. 7(b)) and non-autoregressive scenarios (Fig. 7(a)) inspired by decoder architecture [28].

Let us start from $M_1^l,\ldots, M_k^l$ resulting from the segment-wise application of VE_0^l (Fig. 5-❶). As discussed, this step has space complexity of $\mathcal{O}\big(k(s+t)^2\big)$ for the SA mechanism. Recall that each $M_i^l[:t]$ corresponds to the $\mathcal{X}$-word embeddings. Instead, each $M_i^l[t:]$ corresponds to token embeddings. Let us rename these sequences:

$$X_i^l = M_i^l[:t],\ S_i^l = M_i^l[t:]$$

$$X^l = concat(X_0^l, \ldots, X_k^l),\ S^l = concat(S_0^l, \ldots, S_k^l)$$

1. In a non-autoregressive scenario. We apply VD^l between $\mathcal{X}$-word (X^l) and sequence embeddings S^l (Fig 7a-❷):

$$\text{VD}^l(X_l, S_l)$$

which has space complexity for the first MHSA layer of $\mathcal{O}(ktn)$.

2. Instead, in an autoregressive scenario. Let $T \in \mathbb{R}^{m\times d}$ be the target sequence of length m. We apply VD^l between T and S^l (Fig. 7b-❷):

$$\text{VD}^l(T, S^l)$$

which has space complexity $\mathcal{O}\big(m^2\big)$ for the first MHSA layer, and $\mathcal{O}(nm)$ for the next one. Note that, If $m << n$ the decoder layers do not have a big impact on the memory. One can still resort to the attention patterns in Fig. 4 if $m \simeq n$.

Space-Time Analysis. To show the sub-quadratic scaling laws for the previous architecture, we need to show that the terms $\mathcal{O}\big(k(s+t)^2\big)$ and $\mathcal{O}(kst)$ are sub-quadratic. For simplicity, let us drop the term t as it is constant, so that $\mathcal{O}\big(k(s+t)^2\big) = \mathcal{O}\big(ks^2\big)$ and $\mathcal{O}(kst) = \mathcal{O}(ks)$. Notice that, $ks = n$, thus $\mathcal{O}(ks) = \mathcal{O}(n)$. Ultimately, the scaling laws depend on how the terms k and s relate to sequence length n. For example, if $k = s = \sqrt{n}$, then $\mathcal{O}\big(ks^2\big) = \mathcal{O}(n\sqrt{n})$. Thus proving the sub-quadratic scaling laws. However, other choices of k and s can further reduce memory requirements. For example, $s = \log n$ and $k = n/s$ gives $\mathcal{O}\big(ks^2\big) = \mathcal{O}(n \log n)$. Instead, when $s = n^{1/a}$ and $k = n/s$, we have $\mathcal{O}\big(ks^2\big) = \mathcal{O}(n\sqrt[a]{n})$. Also, we have $\mathcal{O}\big(ks^2\big) = \mathcal{O}\big(n^2\big)$ if $s = n$ (thus $k = 1$) restoring the original quadratic law. In this scenario, the CombTransformer architectures collapse to the Transformer architecture with memory embedding [42]. If one also removes $\mathcal{X}$-word embeddings, by setting $t = 0$, then the CombTransformer architectures collapse to the traditional Transformer architecture. On the other hand, if we set $s = 1$ (thus $k = n$) then we have $\mathcal{O}\big(ks^2\big) = \mathcal{O}(n)$ and $\mathcal{O}(ks) = \mathcal{O}(n)$.

Similar considerations also apply to time analysis. For instance, consider the time complexity of the SA layer, which is $\mathcal{O}\big(n^2 + n^2 d\big)$ (where d is the embedding size). The CombTransformer architectures involve the repetitive application of the SA mechanism on segments, with k segments of size s. This results in a time complexity of $\mathcal{O}\big(ks^2 + s^2 d\big)$. Just like in the space case, the time complexity is influenced by the relationship between k and s. For example, if s is constant, the time complexity becomes $\mathcal{O}(nd)$. If $k = s = \sqrt{n}$, the time complexity becomes $\mathcal{O}(n\sqrt{n} + dn\sqrt{n})$.

In essence, the selections of k and s are contingent on the specific task and the permissible memory allocation for attention. Depending on the situation, there are advantageous scenarios. For instance, processing a few lengthy segments (with a small k and large s) might be favorable in certain cases. Conversely, processing numerous short segments (with a large k and small s) might be more effective in other situations. It is worth noting that as s increases, the scaling laws tend to become more significant. In our assessment, we opted to set $s = k = \sqrt{n}$ in order to strike a harmonious equilibrium between the number of segments and their individual lengths, all while maintaining a sub-quadratic memory requirement.

Nevertheless, the selection of both the model variant and the values for k and s can be approached by treating attention patterns (1, 2, and 3) and architectures (FCT, EDCT, and HCT) as hyperparameters. These hyperparameters could be determined through an initial step of hyperparameter optimization.

V. Evaluation

In this section, we will evaluate the CombTransformer compared to other baselines with different datasets.

Hardware setup. All models have been trained on a single NVIDIA RTX 3090 GPU with a 12th Gen. Intel CPU i9-12900KF, and 128Gb of available RAM.

Hyper-parameters. To fairly evaluate all baselines, we fine-tuned all baselines separately on the dataset java-small using ASHAScheduler [41], and tree-structured Parzen estimators [43] algorithms. We used the implementation provided by the raytune[1] package. Each baseline was fine-tuned for two epochs on the train split of java-small and for ten trials. The optimized hyper-parameters include dropout, learning rate, Adam beta1 and beta2 [44], activation function, and architecture-specific hyper-parameters such as window size (Longformer), chi-word embedding (CombTransformer), relative attention n.

[1]https://www.ray.io/

buckets(T5), relative attention max distance.[2] Apart from fine-tuned hyper-parameters, others are chosen using popular values. E.g., all architectures share the embedding size (128), the number of SA heads (4), and the training batch size (100). Finally, we trained all architectures with Adam optimizer [44].

A. Method Name Generation

Introduction. To summarize code snippets is one of the first steps in code-understanding systems. Such systems can be useful to automatically produce documentation or to categorize code bases. The availability of big data renders these tasks also good benchmarks for neural architectures.

Task. Given a method with its name redacted, the model is asked to generate a token sequence representing the redacted name.

Metrics. We adopt the same metrics used in [8] and [9]:

- true-positives (TP). The number of sub-tokens predicted that are present in the original name.
- false-positives (FP). The number of sub-tokens predicted that are not present in the original name.
- false-negatives (FN). The number of sub-tokens in the original name that are not present in the prediction.

Using TP, FP, and FN, one can compute:

$$\underbrace{\frac{TP}{TP+FP}}_{\text{precision}} \quad \underbrace{\frac{TP}{TP+FN}}_{\text{recall}} \quad \underbrace{\frac{2 * precision * recall}{precision + recall}}_{\text{f1-score}}$$

Both precision, recall, and f1-score are normalized scores ranging from 0 to 1; the best score is 1.

Dataset. The dataset (java-large[3]) is the same used in [8], [9]. It consists of 9,550 Java projects for around 16M Java methods. Training, validation and test splits are made of 9,000, 250 and 350 projects respectively. We parsed java-large with `JavaParser` (v3.25.2) [35]. We obtained five views of java-large from five different processing pipelines:

- c2v view: it is composed of leaf-to-leaf paths extracted from the method AST as in [8].
- c2s view: it is composed of leaf-to-leaf paths extracted from the method AST as in [9].
- f2v view: it is composed of a list of statement-level features as in [20].
- raw view: it is composed of tokenized plain text. We used a Byte-Pair Encodings [45] tokenizer trained using the HuggingFace tokenizers[4] library.
- f2s view: it is a code representation obtained from the unfolding of statement as discussed in Section III.

Models. Beyond the CombTransformer architectures, we trained three non-Transformer and four Transformer-based architectures from the literature as baselines:

- *Code2vec* [8]. It uses bags of leaf-to-leaf AST paths to represent methods. It uses a GA mechanism to encode bag of paths into fixed-size embedding which. Code2vec is trained on the c2v view of java-large.
- *Fold2vec* [20]. It uses TUs as code representation. The architecture uses both LSTM and GA to encode statements. Statement encodings are processed using an SA. Fold2vec is trained on the f2v view of java-large.
- *Code2seq* [9]. It uses bags of leaf-to-leaf AST paths to represent methods. The architecture is an enhancement of *Code2vec*. It is trained on the c2s view of java-large.
- *Transformer* [28]. It is a popular all-attention architecture. It uses SA, fully connected and normalization layers. It is trained on the raw view of java-large.
- *Roberta* [22]. It is a SA-based architecture. It successfully applied to code-related tasks with CodeBERT [46]. Its architecture is trained on the raw view of java-large.
- *Longformer* [31]. It is a variant of the Transformer architecture with a reduced memory cost of $\mathcal{O}(ln)$. Where tokens can only attend to tokens inside a sliding window of size l. It is trained on the raw view of java-large.
- *T5* [47]. It is a SA-based architecture trained on multiple tasks simultaneously, here, limited to the name generation task. T5 is trained on the raw view of java-large.
- *Funnel CombTransformer* (FCT). It is discussed in Section IV. It is evaluated with all proposed attention patterns: FCT-1 (Fig. 4(b)), FCT-2 (Fig. 4(c)), and FCT-3 (Fig. 4(d)). FCTs are trained on the f2s view of java-large.
- *Hourglass CombTransformer* (HCT). It is discussed in Section IV. It is evaluated with all proposed attention patterns: HCT-1 (Fig. 4(b)), HCT-2 (Fig. 4(c)), and HCT-3 (Fig. 4(d)). HCTs are trained on the f2s view of java-large.
- *Encoder-Decoder CombTransformer* (EDCT). It is discussed in Section IV. It is evaluated with all the proposed attention patterns: EDCT-1 (Fig. 4(b)), EDCT-2 (Fig. 4(c)), and EDCT-3 (Fig. 4(d)). EDCTs are trained on the f2s view of java-large.

Apart from Code2vec and Fold2vec, all other architectures are autoregressive. All baselines are trained using the cross-entropy loss with a batch size of 100 and a maximum sequence length of 512. All autoregressive architectures use three encoder and three decoder layers. This allows for lean architectures that can be trained in a few hours (circa 9 hours). We trained all architectures from scratch for 5 epochs (without pretraining) to have a fair comparison between architectures. However, pretraining with goals as masked language modeling may give better results [23].

Results. Table I summarizes the results. The best performance is achieved by the HCT-1 architecture, followed by HCT-3 and HCT-2. These architectures achieve similar results to each other (around 0.60 of f1-score). Slightly worse performances are achieved by FCT and EDCT architectures which respectively obtain around 0.57 and 0.58 of f1-score. The best baseline among the Transformer-like architecture is the Longformer, achieving 0.54 f1-score. During inference, the fastest architecture is Code2vec. Being shallow architectures, both Code2vec, Code2seq, and Fold2vec are able to process thousands of samples per second on GPU. Note that, FCT-1 architecture is able to achieve a considerable inference speed. Firstly, notice that the scaling laws of the SA suggest that it is more efficient (in terms of memory and time) to process many short sequences

[2] https://huggingface.co/docs/transformers/v4.29.1/en/model_doc/t5#transformers.T5Config

[3] https://s3.amazonaws.com/code2seq/datasets/java-large.tar.gz

[4] https://huggingface.co/docs/tokenizers/index

TABLE I
BASELINES RESULTS FOR METHOD NAME GENERATION TASK. THE SMALLER THE BETTER FOR PARAMS., THE HIGHER THE BETTER FOR THE REMAINING METRICS

Model	Params. (M)	pr.	re.	f1	CPU (samples/s)	GPU (samples/s)
code2seq	41	0.576	0.459	0.511	66.456	2313.591
code2vec	311	0.429	0.332	0.374	124.345	2950.531
fold2vec	40	0.599	0.495	0.542	63.227	1277.409
EDCT-1	22	0.625	0.539	0.579	40.656	1168.793
EDCT-2	22	0.64	0.537	0.584	19.896	586.101
EDCT-3	22	0.636	0.537	0.582	19.911	587.57
FCT-1	22	0.605	0.551	0.577	104.321	1848.53
FCT-2	22	0.597	0.536	0.565	31.713	822.703
FCT-3	22	0.604	0.537	0.569	25.942	717.938
HCT-1	23	0.637	0.574	0.603	22.605	642.438
HCT-2	23	0.633	0.562	0.595	11.972	363.933
HCT-3	23	0.636	0.561	0.596	11.041	332.251
longformer	21	0.595	0.505	0.546	23.965	866.517
roberta	21	0.533	0.416	0.467	16.606	563.757
t5	34	0.56	0.456	0.503	15.035	571.25
transformer	22	0.583	0.476	0.524	15.401	481.122

TABLE II
BASELINES RESULTS FOR CODE SEARCH TASK. THE SMALLER THE BETTER FOR PARAMS., THE HIGHER THE BETTER FOR THE REMAINING METRICS

Model	Params. (M)	Top-1	MRR	CPU (samples/s)	GPU (samples/s)
code2seq	28	0.468	0.519	157.257	7120.265
code2vec	298	0.264	0.322	279.786	10099.636
fold2vec	27	0.56	0.601	140.394	5123.235
EDCT-1	29	0.538	0.623	188.36	6242.339
EDCT-2	29	0.564	0.647	120.097	3818.302
EDCT-3	29	0.563	0.642	106.525	3405.643
FCT-1	28	0.567	0.657	395.496	13006.912
FCT-2	28	0.489	0.579	185.722	6054.496
FCT-3	28	0.463	0.557	162.429	5355.325
HCT-1	30	0.533	0.621	111.129	3778.538
HCT-2	31	0.536	0.621	72.813	2350.49
HCT-3	31	0.528	0.621	67.299	2184.446
longformer	27	0.554	0.618	149.832	4918.995
roberta	27	0.52	0.555	176.49	6222.663
t5	53	0.55	0.62	153.297	5410.793
transformer	27	0.542	0.584	172.478	6227.57

than a few long ones. FCT-1 exploits this fact, by processing many segments rather than entire sequences. Compared to FCT-1, other architecture add complexity by either using more complex attention patterns or by introducing more complex layers. For this reason, they achieve slower inference speeds. In terms of parameters, apart from Code2vec, all architectures have less than 50M trainable parameters. Code2vec maintains a very large embedding table that accounts for the majority of leaf-to-leaf path hashes. While the elevated number of parameters and its shallow nature allow for fast inference times.

B. Code Search

Introduction. The programming activity involves several tasks. Among them, the ability to efficiently search for useful and relevant code snippets is essential. Search engines capable of proposing relevant code snippets, given natural language queries, are tools capable of boosting the programmer productivity. To fairly evaluate our proposal, we follow the procedure described in [15].

Task. Given a natural language query, the model looks a codebase for relevant snippets. To learn this task, the model is trained to match documentation snippets with their implementation. In practice, the model is presented with a batch of several methods and their corresponding documentation strings. The model is trained to match code snippets to the associated natural language query.

Metrics. We evaluate the proposed architecture on:

- Top-1 accuracy (top-1). This represents the number of correct associations over the total number of queries.
- Mean Reciprocal Rank (MRR). It is the mean of the reciprocal rank of all queries:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$$

where, N is the number of all queries, $rank_i$ is the rank of the correct answer wrt. the query q_i.

Dataset. To train our models, we used the Java portion of the CodeSearchNet dataset [15]. The training split contains $454,450$ samples. The validation splits contain $15,327$ samples. Lastly, the test split of CodeSearchNet contains $26,908$ samples. Every sample of CodeSearchNet is composed of 1. a Java method and 2. the documentation string attached to the method.

As before the dataset is used to generate several views depending on the processing required by the different architectures: c2v/c2s/f2v/f2s/raw views.

Models. We adapted and retrained the previously presented 16 architectures for the code search task. All models are trained with a sequence of 256 tokens (both for query sequences and code sequences). All models are trained with the same batch size of 100 for 50 epochs. Apart from Code2vec, Code2seq, and Fold2vec, all models have the same vocabulary of $100,000$ query tokens and $100,000$ for code tokens. Instead, Code2vec, Code2seq, and Fold2vec retain their original vocabulary size. Again, apart from Code2vec, Code2seq, and Fold2vec, all architectures are composed of 3 encoder layers for method snippets and 3 encoder layers for documentation. Additionally, all the models are trained with the same attention loss:

$$\mathcal{L}(Q) = -\frac{1}{N}\sum_{i}\log\frac{\exp(E_c(c_i)^T E_q(q_i))}{\sum_j \exp(E_c(c_j)^T E_q(q_i))}$$

where q_i represents the i-th natural language query, c_i represents the correct associated method body, Q is the set of all samples in a batch. E_c and E_q represent the code encoder and the query encoder respectively. When $j \neq i$, c_j is an incorrect answer for the query q_i (also called distractors). By minimizing the loss, we maximize the inner product between q_i and c_i. Instead, when $i \neq j$, the loss minimizes the inner product between q_i and c_j.

Results. Table II summarizes the results. The best-performing model is FCT-1 (achieving an MRR of 0.657). Most notably, other FCT architectures achieve considerably lower results. This is probably caused by the fact that cross-segment attention introduces complexity to the information flow of the SA layers, which in turn produces a code embedding that is drastically more difficult to align with the documentation embedding. The contrary happens when considering EDCT architectures. EDCT architectures, being endowed with an internal decoder layer, have the capacity to align code $\mathcal{X}$-word

TABLE III
BASELINES RESULTS FOR THE SUMMARIZATION TASK. THE SMALLER THE BETTER FOR PARAMS. THE HIGHER THE BETTER FOR ALL THE REMAINING METRICS

Model	Params. (M)	f1	bleu	CPU (samples/s)	GPU (samples/s)
code2seq	41	0.557	0.102	1.943	96.345
code2vec	311	0.516	0.108	2.655	112.671
fold2vec	40	0.565	0.098	2.032	74.43
EDCT-1	41	0.56	0.129	1.959	81.597
EDCT-2	41	0.562	0.149	1.668	68.937
EDCT-3	41	0.564	0.144	1.576	64.954
FCT-1	40	0.558	0.114	2.332	89.983
FCT-2	40	0.563	0.124	1.944	78.002
FCT-3	40	0.564	0.129	1.846	74.56
HCT-1	42	0.572	0.141	1.599	60.437
HCT-2	42	0.572	0.144	1.282	50.216
HCT-3	42	0.571	0.152	1.246	48.04
longformer	40	0.569	0.142	1.676	73.429
roberta	40	0.563	0.131	1.738	73.323
t5	53	0.556	0.134	1.782	75.964
transformer	40	0.568	0.128	1.549	63.317

embedding with their documentation counterparts when cross-segment attention is allowed. Instead, HCT architectures are unaffected by the issue. As the HCT layer uses skip connections to incrementally modify the input embeddings, the resulting architecture is able to align the resulting code embedding and the documentation embedding regardless of the attention pattern used. Among Transformer-like architectures, the best-performing model is T5 achieving an MRR of 0.62. Regarding inference speeds, Code2vec, and Code2seq, due to their shallow nature, are able to process $10,000$ samples per second. This kind of speed is matched only by FCT-1. Overall, all models are several times faster compared to the previous task. This behavior can be explained by the fact that wrt. the previous task, models were required to make eight prediction (one for each target token) to complete a batch.

C. Code Summarization

Introduction. Code summarization represents one of the main applications for deep learning architectures, as it can speed up the generation of documentation. Compared to the previous task, code summarization is more difficult as it requires a deeper understanding of the code snippets and the ability to generate coherent natural language descriptions.

Task. Given a code snippet, the model generates a code description. To learn this task, the model is trained in an autoregressive manner to match a given description.

Metrics. Since architectures must generate long sequences, the previous metrics (precision, recall, and f1) cannot capture the quality of the generated output. We use the BLEU score [48]. It is a quality metric based on the co-occurrences of n-grams between the target and the predicted sequence. We use the implementation by the TorchMetrics[5] Python package with 4-grams and with smoothing.

Dataset. We used the Funcom dataset proposed in [49]. Funcom was applied to recent studies [24], [50]. Funcom is composed of 2M of Java methods alongside their documentation. We applied a training, validation, and test split of 70%, 10%, 20% respectively. As before, Funcom is used to generate several views (c2v, c2s, f2v, f2s, and raw) depending on the processing required by the different architectures.

Models. We adapted and trained all the previously mentioned architectures. All the models can process a sequence of length up to 256 tokens and can generate documentation of length up to 64 tokens (same settings of [46]). All architectures are trained with an embedding size of 128 and a vocabulary size of $100,000$. Additionally, all architectures are composed of 3 encoder layers and 3 decoder layers for the token generation. All architectures are trained with Adam optimizer [44] and hyperparameter obtained from the previous fine-tuning for 10 epochs. Finally, all architectures are trained with cross-entropy loss as in Section V-A.

Results. HCT-3 is the architecture that achieves the best results (0.152 BLEU). For the summarization task, being the most difficult, allowing cross-segment attention produces improvement in all the proposed architectures. Among the proposed models, the FCT groups perform slightly worse. Meaning that compressing all the embeddings in a few ones does not allow the necessary information to pass through the last layers of the architecture. Similarly, all the Transformer-based architectures perform achieve around 0.13 of BLEU score with the exception of the Longformer that achieves BLEU score of 0.142. Instead, Code2seq, Code2vec, and Fold2vec fall slightly behind (BLEU score of around 0.1). Their shallow nature does not allow them to get results on par with deeper models. As to inference speed, the fact that each prediction, to be completed, needs 64 evaluation of the model (one per token) causes considerably slower models. Even shallower architectures are not particularly faster wrt. others as the inference time are dominated by the last decoder layers performing the generation.

VI. DISCUSSION

$\mathcal{X}$-word embedding In this section, we discuss a few of the interesting behaviors for $\mathcal{X}$-words embeddings. The following experiments are performed on a FCT-1.

A $\mathcal{X}$ can be seen as a vector having an *attention budget* to spend between all input tokens. This budget is used to combine input tokens into a summary embedding through a weighted sum. This vector is forced to put its budget into relevant embeddings in order to have relevant information in successive layers. If we use t $\mathcal{X}$-word embeddings, we will have t summary embeddings for each segment.

Fig. 8 ② shows a parsed representation of the code snippet in ① - 1 circle. Here, all the tokens that take the maximum attention budget from at least one $\mathcal{X}$ are highlighted. Note that less than 7 tokens are often highlighted. Fig. 8 (❶ and ❷) shows how all 7$\mathcal{X}$s spend their attention budget among the tokens of the third and fifth statements. Note that, Different $\mathcal{X}$s spend their budgets differently. This highlights that each $\mathcal{X}$ specializes to match a specific type of token. Although $\mathcal{X}$s spend their budget differently, they often also attend to the same tokens. This fact shows that not all tokens are considered equally important. Those tokens that attend the most wrt. at least one $\mathcal{X}$s can be seen as keywords for the respective statement.

[5]https://torchmetrics.readthedocs.io/en/stable/text/bleu_score.html

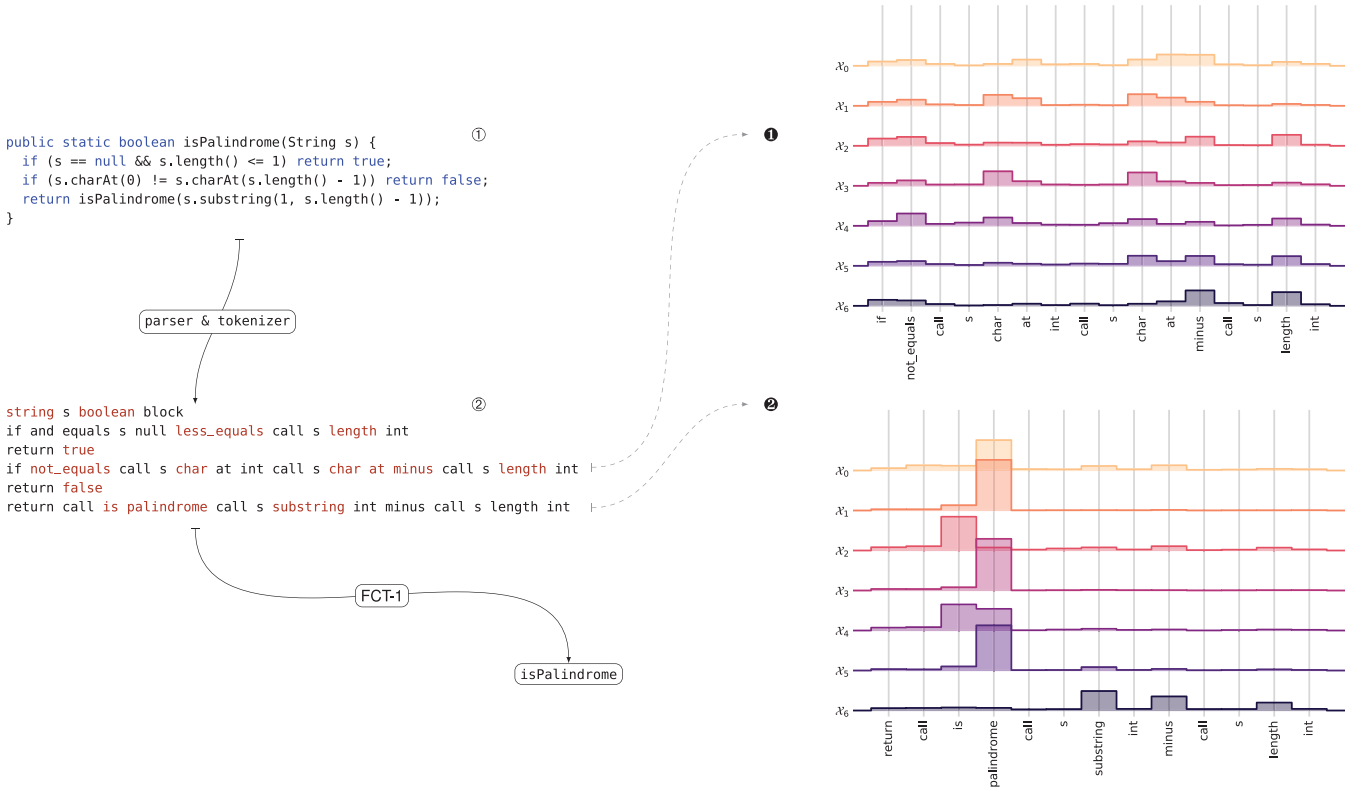


Fig. 8. Distribution of $\mathcal{X}$s attention between tokens. Panel ① shows a Java method, panel ② the parsed code, as discussed in Section III. Colored tokens received max attention from at least one $\mathcal{X}$s. Panels ❶ and ❷ show how $\mathcal{X}$s attend to tokens from the first and second statement respectively.

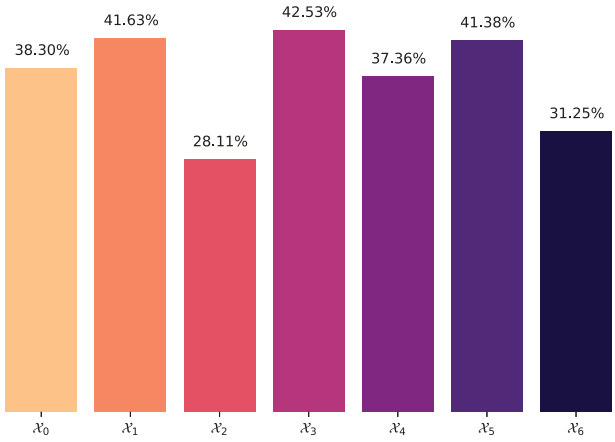


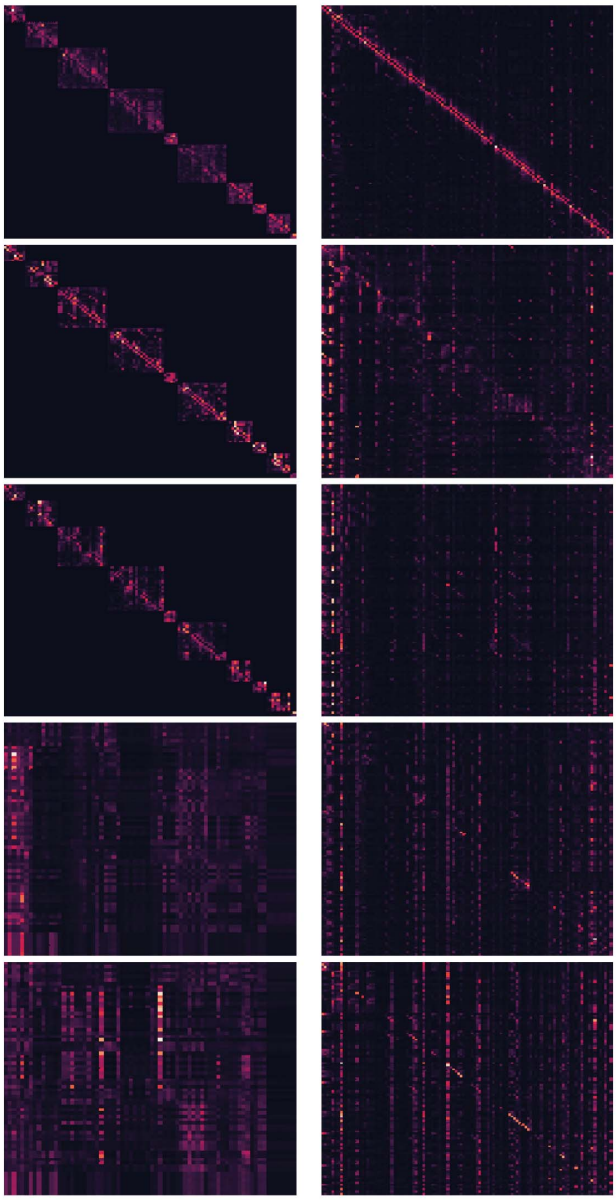
Fig. 9. Attention budget given to tokens appearing in FCT-1 prediction.

We argue that forcing the network to compress a sequence into a few embeddings forces the network to pick the most informative tokens, like keywords for a natural language sentence. Meanwhile, tokens that receive a low attention budget are dropped and not further processed. This concept of compressing information is used in several types of architecture across several fields. For example, the funnel layer in [37] for NLP, or resnet architectures [40] in computer vision.

Next, we investigate how $\mathcal{X}$s spend their budget. Fig. 9 shows how each $\mathcal{X}$s attend to tokens that appear also in the prediction. For example, when predicted tokens appear in a statement, $\mathcal{X}_0$ spends 38.30% of his attention budged on these tokens (these statistics are computed from the test set). Since the network is trained to predict target tokens, this highlights how much target tokens become relevant for the CombTransformer. This should be even more evident when one considers that 61.45% of target tokens (from the test set) also appear in the method body. Thus, it should not surprise that $\mathcal{X}$s put a lot of their attention into these tokens.

These last considerations are based on crude approximations. We discussed the attention given to tokens. However, to be truly precise, $\mathcal{X}$-word embeddings attention is not given directly to tokens but rather to their embeddings after a certain number of encoding steps (see Fig. 5-❸). Thus, attention that is given to a certain token depends also on other tokens from the same statement.

Following the work of previous studies [42], [51], we are going to show the main differences between the CombTransformer attention and the Transformer attention. In particular, we are going to compare the mean between different attention heads for each encoder layer of a FCT-1 and a Transformer model. Fig. 10(a) displays 5 heatmaps, one for each encoder of the funnel Transformer. The first three are layers in the teeth. The last two are layers in the backbone. In the backbone, $\mathcal{X}$s of different teeth attend to each other. Similarly, Fig. 10(b) displays 5 heatmaps, one for each encoder layer of the Transformer model. The first encoder in Fig. 10(b) shows a predominant diagonal pattern which is substituted by column patterns in later layers. A diagonal pattern almost identical arises in the second encoder of Fig. 10(a). This behavior indicates that the FCT-1 is somewhat successfully approximating the behavior of



(a) CombTransformer attention (b) Transformer attention

Fig. 10. Attention heatmaps for a FCT-1 model (left) and the Transformer model (right). All heatmaps are obtained from the same sample. Each heatmap represents the mean between heads of the MHSA matrices ($softmax(Q \cdot K^T/d)$ in Section II).

the Transformer model while using less space. Interestingly, the last two encoders in Fig. 10(a) present patterns which are completely different from those of the Transformer. Not only, but these patterns are different from those found in [51]. This suggests that computation diverges drastically. This shows that $\mathcal{X}$s interact with each other in a unique way, different from normal token embeddings.

A. Representation

The CombTransformer baselines are not bounded by the proposed representation. In this section, we investigate the effects of the proposed code representation. Table IV summarizes the results. For both architecture, using the proposed representation has beneficial effects. In particular, for the HCT-1 architecture there is noticeable improvement of the 8 in terms of f1-score. Similarly, the Transformer architecture improves of 5. This shows that using parsed representation of source code can have beneficial effects regardless of the network architecture. Fig. 11 shows the f1-score obtained during validation for the HCT-1 and the Transformer architectures. In both scenarios, the effect of the new representation is noticeably positive.

TABLE IV
PERFORMANCE FOR THE TRANSFORMER AND THE HCT-1 ARCHITECTURE UNDER DIFFERENCE DATA REPRESENTATION

Model	Data view	Params.(M)	pr.	re.	f1
HCT-1	raw	23	0.57	0.473	0.517
HCT-1	f2s	23	0.636	0.561	0.596
transformer	raw	22	0.583	0.476	0.524
transformer	f2s	22	0.615	0.534	0.572

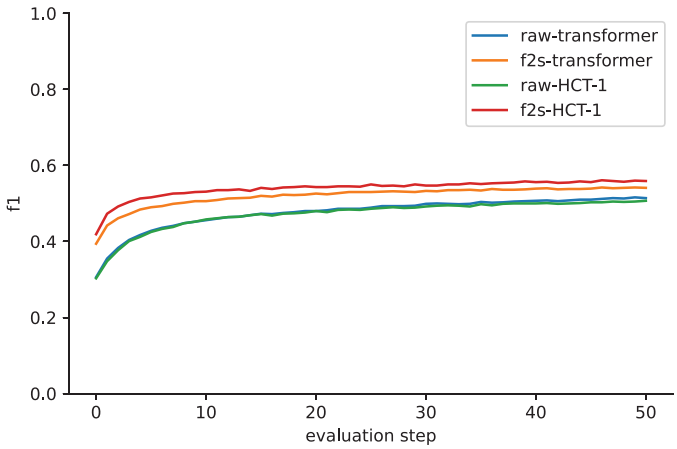


Fig. 11. f1 scores during validation after each $1,000$ training batches (1M data points) for 10 epochs.

B. Memory Scaling

As mentioned earlier, the memory scaling laws of SA are quadratic ($\mathcal{O}(n^2)$) in the sequence length (n). Meanwhile, the scaling laws of the CombTransformer architecture depend on the relation between the number of segments (k) and the segment lengths (s). In this section, we compare the scaling laws with the GPU memory used to process samples. Firstly, consider Fig. 12(a), here we compare the theoretical memory requirements of the single-head SA mechanism wrt. a Pytorch implementation of single-head SA. Most notably, the memory requirements grow rapidly beyond the classical single GPU requirements (16GB). As shown in Fig. 12(a) the orange lines are stopped early as the required memory grows past the available memory. Moreover, if one accounts also for embedding size, multiple attention heads, multiple layers, batch size, and gradient buffers the maximum sequence that can be processed on a single GPU with 16GB is much smaller. Instead, by processing sequence segments separately, it is possible to shrink the memory requirements of the SA layer (shown in Fig. 12(b) and (c)). This effectively frees GPU memory that

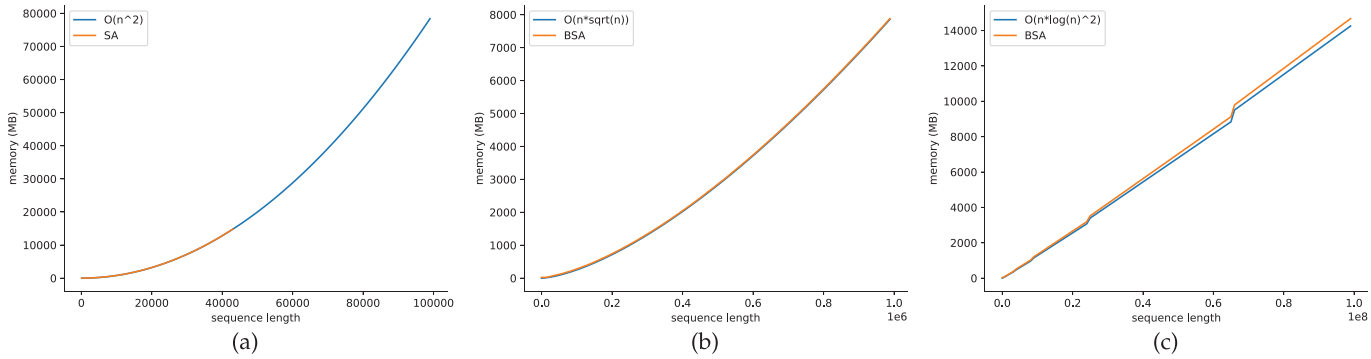


Fig. 12. Memory scaling for the SA mechanism wrt. their theoretical Memory requirements. Fig. 12(a) shows the classical SA mechanism. Fig. 12(b) shows the SA mechanism with segments when $k = \sqrt{n}$, and $s = \sqrt{n}$. Fig. 12(c) shows the SA mechanism with segments when $k = n/\log n$ and $s = \log n$.

can be used to either process longer sequences or stack more layers. E.g., when $k = \sqrt{n}$ and $s = \sqrt{n}$, as shown in Fig. 12(b), the SA mechanism can be applied on much longer sequences.

VII. Threats to Validity

Now, we discuss threats that may have affected our results.

Conclusion validity. We claimed that the CombTransformers brings an improvement over the Transformer. However, the improvement wrt. the Transformer is marginal. One could question whether the improvement is effective or it has come by chance. To mitigate this issue, we evaluated few variants of the comb Transformers (FCT, HCT, and EDCT each with different attention patterns). We also compare other popular efficient Transformer, Longformer. While it is not feasible to train several models for each architecture due to training time, we are confident that the quantity of used baselines are enough to minimize any bias.

Internal validity. Several methods proposed in the literature compress the SA mechanism. Such as [52], [53], [54], [55], [56] and many others. Unfortunately, evaluating all these approaches with our current resources is infeasible. In this respect, we chose to compare our architecture with the Longformer on the same tasks. Longformer architecture is one among the best performing architectures in Tay et al.'s study [57]. We used a third-party implementation for Longformer (both freely accessible [58], [59]) to limit any bias we could have introduced by implementing them ourselves. During the development of the CombTransformers, there were a lot of architectural changes and manual parameter tuning. However, the same process was not applied for the Transformer and Longformer to keep the baselines fair to the originals. We cannot exclude that this has affected the results. To mitigate this issue, we have included among other baselines, models that were specifically designed for this task (Code2seq and Code2vec).

External validity. We claimed that the shown improvements wrt. Code2seq come from the representation and the model. However, Code2seq and CombTransformer employ different preprocessing procedures yielding slightly different datasets. However, we kept the preprocessing as close as possible to those of the previous works while using a different tokenizer and an updated parser.

VIII. Related Works

Let us overview some approaches that shares either goals, technique or application domain with CombTransformer.

Memory Efficient Transformer. A lot of work is spent trying to reduce the memory footprint of Transformers similarly to what the CombTransformer is trying. In Roy et al. [52] a k-means clustering algorithm is used to cluster queries and keys. Then, the attention mechanism is performed only on queries and keys from the same cluster. A similar approach using locality sensitive hashing is used in Kitaev et al. [55]. In Beltagy et al. [31] tokens can only attend to other close tokens. The same concept is then expanded in [53] with random attention. Instead an approximation for softmax attention is applied in Choromanski et al. [60]. A similar technique to ours is applied in Dai et al. [37] where inner Transformer layers compress the sequence size using pooling. A similar segmenting approach of statements is used by Yang et al. [61] in the NLP domain, however the sequence size reduction uses always the same special token.

Model Compression. These techniques aim to reduce the size of trained models without compromising the model performance. Compressed models are leaner and run faster than the original versions. Li et al. [62] presents a weight pruning techniques to reduce the number of weights in convolutional NN, whole layers are pruned [63] providing leaner models. A different approach lies in knowledge distillation. Here a pretrained model—teacher—is used to transfer the knowledge to a usually smaller model—student. Huang et al. [64] proposes to align inter-class relation between the teacher model and the student model predictions. Another popular technique is the model quantization. Here, the number of bit available for each weight are reduced. Jacob et al. [65] proposes a quantization scheme that allows NN to be run using only integer arithmetic.

Machine Learning on Code. Allamanis et al. [19] proposed the *extreme source code summarization* task. Allamanis et al. [66] proposed a convolutional NN to predict method names from source code. More successful approaches were applied by Alon et al. [7], [8], [9] which we have already discussed. Xu

et al. [67] proposed an approach based on hierarchical attention. For the task of *code summarization*, Movshovitz-Attias et al. [68] proposed to use comment to summarize code. Once again, an attention based NN was proposed by Iyer et al. [69]. A more successful and recent approach, based on Transformers architecture, was proposed by Feng et al. [46]. *Automatic bug detection* may be an application for the CombTransformer; notable examples: Shi et al. [70] and Dam et al. [13]. The former applies an approach close to Code2seq, the latter uses to Tree-LSTM from the AST. Similarly, Tufano et al. [71] applies a NMT architecture to automatically generate patches for buggy code.

IX. Conclusion

In this work, we designed some novel architectures (CombTransformers) based on the Transformer. The comb Transformers are all designed to deal with long sequences, specifically when these sequences can be seen as a sequence of sentences (e.g., methods are a sequence of statements). Comb Transformers effectively reduce the memory requirements enabling the training on longer sequences. They also achieve comparable results wrt. established architectures as the Longformer and the Transformer. There is no reason to prevent a CombTransformer architecture to be used in code-unrelated tasks. Different domains could benefit from this kind of architectures. This will be our future investigation.

Lastly, we provide a package to reproduce our experiments which is available at the following address:

https://cazzola.di.unimi.it/comb-transformer.html

References

[1] K. Ellis et al., "DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning," in *Proc. PLDI*, Jun. 2021, pp. 835–850.

[2] M. Amodio, S. Chaudhuri, and T. Reps, "Neural attribute machines for programming generation," May 2017, *arXiv:1705.09231*.

[3] V. Murali, L. Qi, S. Chaudhuri, and C. Jermain, "Neural sketch learning for conditional program generation," in *Proc. ICLR*, May 2018.

[4] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. PLDI*. New York, NY, USA: ACM, Jun. 2014, pp. 419–428.

[5] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proc. ICML*. PMLR, Jul. 2020.

[6] M. Chen et al., "Evaluating large language models trained on code," Jul. 2021, *arXiv:2107.03374*.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proc. PLDI*, Philadelphia, PA, USA. New York, NY, USA: ACM, Jun. 2018, pp. 404–419.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. POPL*, Cascais, Portugal. New York, NY, USA: ACM, Jan. 2019, pp. 40:1–40:29.

[9] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. ICLR*, 2019.

[10] A. Celik, P. Sreepathi, S. Khurshid, and M. Gligoric, "Bounded exhaustive test-input generation on GPUs," in *Proc. OOPSLA*, Oct. 2017, pp. 94:1–94:25.

[11] A. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," in *Proc. OOPSLA*, Oct. 2017, pp. 93:1–93:29.

[12] J. He, C.-C. Lee, V. Raychev, and M. Vechev, "Learning to find naming issues with big code and small supervision," in *Proc. PLDI*, pp. 296–311.

[13] H. K. Dam et al., "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proc. MSR*, Montréal, QB, Canada. Piscataway, NJ, USA: IEEE, May 2019, pp. 46–57.

[14] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," in *Proc. OOPSLA*, Oct. 2018, pp. 147:1–147:25.

[15] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.

[16] C. Wu and M. Yan, "Learning deep semantic model for code search using CodeSearchNet corpus," 2022, *arXiv:2201.11313v1*.

[17] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison–Wesley, Jun. 1999.

[18] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proc. ECOOP*, Genoa, Italy. Springer, Jun. 2009, pp. 294–317.

[19] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proc. FSE*, Sep. 2015, pp. 38–49.

[20] F. Bertolotti and W. Cazzola, "Fold2Vec: Towards a statement based representation of code for code comprehension," *Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 6:1–6:31, Feb. 2023.

[21] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. ICSE*, Gothenburg, Sweden. New York, NY, USA: ACM, May/Jun. 2018, pp. 933–944.

[22] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," Jul. 2019, *arXiv:1907.11692v1*.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, Jun. 2019, pp. 4171–4186.

[24] A. Mastropaolo et al., "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proc. ICSE*, Madrid, Spain. Piscataway, NJ, USA: IEEE, May 2021, pp. 336–347.

[25] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation," in *Proc. EMNLP*, 2021, pp. 8696–8708.

[26] Q. Liu, M. J. Kusner, and P. Blunsom, "A survey on contextual embeddings," Apr. 2020, *arxiv:2003.07278v2*.

[27] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proc. EMNLP*.

[28] A. Vaswani et al., "Attention is all you need," in *Proc. NIPS*, Dec. 2017, pp. 6000–6010.

[29] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations," in *Proc. ICLR*, Apr. 2020.

[30] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019, *arXiv:1904.10509*.

[31] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," Dec. 2021, *arXiv:2004.05150v2*, pp. 1–17.

[32] M. Hucka, "Spiral: Splitters for identifiers in source code files," *J. Open Source Softw.*, vol. 3, no. 24, pp. 653:1–653:3, Apr. 2018.

[33] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vujay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Softw. Eng.*, vol. 19, no. 6, pp. 1754–1780, Dec. 2014.

[34] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *Proc. ECOOP*, 2011, pp. 130–154.

[35] N. Smith, D. van Bruggen, and F. Tomasetti, *javaParser: Visited*. LeanPub, 2019.

[36] M. S. Burtsev, Y. Kuratov, A. Peganov, and G. Sapunov, "Memory transformer," Jun. 2020, *arXiv:2006.11527*.

[37] Z. Dai, G. Lai, Y. Yang, and Q. V. Le, "Funnel-transformer: Filtering out sequential redundancy for efficient language processing," in *Proc. NeurIPS*, Vancouver, Canada, Dec. 2020, pp. 4271–4282.

[38] I. Melekhov, J. Ylioinas, J. Kannala, and E. Rahtu, "Image-based localization using hourglass networks," 2017, *arXiv:1703.07971v3*.

[39] J. Yang, Q. Liu, and K. Zhang, "Stacked hourglass network for robust landmark localisation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jul. 2017, pp. 2025–2033.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[41] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Proc. NeurIPS*, Dec. 2018.

[42] M. S. Burtsev, Y. Kuratov, A. Peganov, and G. V. Sapunov, "Memory transformer," Feb. 2021, *arXiv:2006.11527v2*, pp. 1–17.

[43] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégi, "Algorithms for hyper-parameter optimization," in *Proc. NIPS*, 2011, pp. 2546–2554.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, San Diego, CA, USA, May 2015.

[45] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. ACL*, 2016, pp. 1715–1725.

[46] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. EMNLP*, 2020, pp. 1536–1547.
[47] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, p. 5485–5551, Jan. 2020.
[48] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. ACL*, 2002, pp. 311–318.
[49] A. LeClair and C. McMillian, "Recommendations for datasets for source code summarization," in *Proc. HLT-NAACL*, Jun. 2019.
[50] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Proc. MSR*, Seoul, South Korea. New York, NY, USA: ACM, Jun. 2020, pp. 300–310.
[51] O. Kovaleva, A. Romanov, A. Rogers, and A. Rumshisky, "Revealing the dark secrets of BERT," in *Proc. EMNLP*, Nov. 2019, pp. 4365–4374.
[52] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, "Efficient content-based sparse attention with routing transformers," *Trans. Assoc. Comput. Ling.*, vol. 9, pp. 53–68, 2021.
[53] M. Zaheer et al., "BigBird: Transformers for longer sequences," in *Proc. NeurIPS*, Vancouver, Canada, Dec. 2020.
[54] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," Jun. 2017, *arXiv:2006.04768v3*.
[55] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *Proc. ICLR*, Addis Ababa, Ethiopia, Apr. 2020.
[56] Y. Tay, D. Bahri, L. Yang, and D. Metzler, and D.-C. Juan, "Sparse Sinkhorn Attention," in *Proc. ICML*, Jul. 2020, pp. 9438–9447.
[57] Y. Tay et al., "Long range arena: A benchmark for efficient transformers," May 2017, *arXiv:2011.04006v1*.
[58] M. Ott et al., "FAIRSEQ: A fast, extensible toolkit for sequence modeling," in *Proc. HLT-NAACL*, Jun. 2019, pp. 48–53.
[59] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. EMNLP*. ACL, Nov. 2020, pp. 38–45.
[60] K. Choromanski et al., "Rethinking attention with performers," in *Proc. ICLR*, Apr. 2021, pp. 1–38.
[61] L. Yang, M. Zhang, C. Li, M. Bendersky, and M. Najork, "Beyond 512 tokens: Siamese multi-depth transformer-based hierarchical encoder for long-form document matching," in *Proc. CIKM*, Oct. 2020, pp. 1725–1734.
[62] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," in *Proc. ICLR*, Toulon, France, Apr. 2017.
[63] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Proc. NeurIPS*, Dec. 2018, pp. 2552–2562.
[64] T. Huang, S. You, F. Wang, C. Qian, and C. Xu, "Knowledge distillation from a stronger teacher," in *Proc. NeurIPS*, Nov. 2022.
[65] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pp. 2704–2713.
[66] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. ICML*, New York, NY, USA. PMLR, Jun. 2016, pp. 2091–2100.
[67] S. Xu, S. Zhang, W. Wang, X. Cao, C. Guo, and J. Xu, "Method name suggestion with hierarchical and attention networks," in *Proc. PEPM*, Cascais, Portugal. New York, NY, USA: ACM, Jan. 2019, pp. 10–21.
[68] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *Proc. ACL*, 2013, pp. 35–40.
[69] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. ACL*, Aug. 2016, pp. 2073–2083.
[70] K. Shi, Y. Lu, J. Chang, and Z. Weu, "PathPair2Vec: An AST path pair-based code representation method for defect prediction," *J. Comput. Lang.*, vol. 59, Aug. 2020.
[71] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw.* Eng. Method., vol. 28, no. 4, pp. 19:1–29, Oct. 2019.

Francesco Bertolotti received the master's degree in computer science from the Università degli Studi di Milano. Previously, he was an Assistant Researcher with the same university, where he is currently working toward the Ph.D. degree in computer science. He is a member of the ADAPT Laboratory. His research interests are programming languages, software quality, and machine/deep learning techniques and its reciprocal cross-fertilization.

Walter Cazzola is currently an Associate Professor with the Department of Computer Science at the Università degli Studi di Milano, Italy, and the Chair of the ADAPT Laboratory. He designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages, and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution, and comprehension, and programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is an Associate Editor for the *Journal of Computer Languages* published by Elsevier. More information about him and all his publications are available at http://cazzola.di.unimi.it.