On-the-Fly Syntax Highlighting: Generalisation and Speed-Ups

Marco Edoardo Palma, Alex Wolf, Pasquale Salza, and Harald C. Gall, *Member, IEEE*

***Abstract*—On-the-fly syntax highlighting involves the rapid association of visual secondary notation with each character of a language derivation. This task has grown in importance due to the widespread use of online software development tools, which frequently display source code and heavily rely on efficient syntax highlighting mechanisms. In this context, resolvers must address three key demands: speed, accuracy, and development costs. Speed constraints are crucial for ensuring usability, providing responsive feedback for end users and minimizing system overhead. At the same time, precise syntax highlighting is essential for improving code comprehension. Achieving such accuracy, however, requires the ability to perform grammatical analysis, even in cases of varying correctness. Additionally, the development costs associated with supporting multiple programming languages pose a significant challenge. The technical challenges in balancing these three aspects explain why developers today experience significantly worse code syntax highlighting online compared to what they have locally. The current state-of-the-art relies on leveraging programming languages' original lexers and parsers to generate syntax highlighting oracles, which are used to train base Recurrent Neural Network models. However, questions of generalisation remain. This paper addresses this gap by extending previous work validation dataset to six mainstream programming languages thus providing a more thorough evaluation. In response to limitations related to evaluation performance and training costs, this work introduces a novel Convolutional Neural Network (CNN) based model, specifically designed to mitigate these issues. Furthermore, this work addresses an area previously unexplored performance gains when deploying such models on GPUs. The evaluation demonstrates that the new CNN-based implementation is significantly faster than existing state-of-the-art methods, while still delivering the same near-perfect accuracy.**

***Index Terms*—Syntax highlighting, neural networks, deep learning, regular expressions.**

I. Introduction

SYNTAX highlighting (SH) is the practice of visually annotating code by associating distinct colours with specific language sub-productions, enhancing code comprehensibility [1], [2]. Code is presented in a multitude of online contexts, such as code review workflows, repository file browsers, and various forms of code snippets, all of which benefit from syntax highlighting (SH) mechanisms [3]. Importantly, these platforms employ SH "On-the-Fly", meaning that SH resolvers compute the highlighting for code just before it is displayed to the user.

The design choice of dynamic SH, driven by space constraints and the inability to cache notations, places SH resolvers in a challenging position. These resolvers must operate efficiently, responding swiftly to a high frequency of requests to ensure platform usability. Furthermore, they are expected to deliver accurate highlighting, correctly associating code sub-productions with their respective SH classes or colours. Achieving this high level of accuracy requires grammatical analysis of the code, though a full parsing process is typically infeasible due to time constraints and the possibility of incorrect derivations [3]. Additionally, addressing development costs remains critical, given the rapidly evolving landscape of mainstream programming languages and their versions. The technical challenges in balancing these three aspects explain why developers today experience significantly worse code syntax highlighting online compared to what they have locally.

Traditionally, developers have relied on manually creating complex systems of regular expressions to achieve SH, an approach that, while effective, is tedious and prone to inaccuracies. More recent efforts have sought to reduce development costs, improve SH accuracy, and expand grammatical coverage. The current *state-of-the-art* approach [3] treats SH as a machine learning translation problem, leveraging the language's original lexer to tokenize the code and bind each token to a SH class. By utilising the language's original parser and lexer to create a SH oracle, this method reduces development costs, requiring only the design of a deterministic Abstract Syntax Tree (AST) walker for each language. This approach excels in accuracy and coverage, including handling incorrect input sequences by making statistically informed SH inferences when final coloring is applied. The efficiency of this solution depends largely on the performance of the underlying machine learning models.

Although the *state-of-the-art* approach has been validated on three mainstream programming languages—*Java*, *Kotlin*, and *Python3* [3]—its generalisation across a broader range of languages remains an open question. This work addresses that question by doubling the size of the validation dataset to six mainstream programming languages. Hence, formal SH models for *JavaScript*, *C++*, and *C#* were developed, and used in the evaluation of the performances of the *state-of-the-art*

Received 30 January 2024; revised 9 September 2024; accepted 11 November 2024. Date of publication 26 November 2024; date of current version 13 February 2025. This work was supported by the Swiss National Science Foundation (SNSF) Project Melise - Machine Learning Assisted Software Development under Grant SNSF204632. Recommended for acceptance by A. Sarma. *(Corresponding author: Marco Edoardo Palma.)*

The authors are with the Department of Informatics, University of Zurich, CH-8006 Zurich, Switzerland (e-mail: marcoepalma@ifi.uzh.ch; wolf@ifi.uzh.ch; salza@ifi.uzh.ch; gall@ifi.uzh.ch).

Digital Object Identifier 10.1109/TSE.2024.3506040

methods. The original approach [3] suggested baseline Recurrent Neural Network (RNN) models, with bidirectional variants Bidirectional Recurrent Neural Network (BRNN) demonstrating the best performance. However, the pursuit of even more efficient models remained an open challenge. In response, this paper introduces a faster Convolutional Neural Network (CNN) model, which is evaluated across all tested languages. The main findings confirm the generalisation of the original approach to the newly evaluated languages while demonstrating how the proposed CNN model maintains accuracy and significantly accelerates computations.

Additionally, this work explores a previously unaddressed aspect of performance: the advantages of running these models on *GPU*s. The analysis highlights the substantial efficiency gains achieved through high-performance hardware, particularly for the novel CNN implementation. Evidence indicates that *GPU*-based evaluation can lead to considerable improvements in the prediction speed of such statistical resolvers, especially for the new CNN model.

In summary, the primary contributions of this paper are:

- An extended SH benchmarking dataset that now includes *JavaScript*, *C++*, and *C#*, doubling the previous dataset of *Java*, *Kotlin*, and *Python*, enhancing the generalisation potential.
- A detailed analysis confirming the generalisation of the *state-of-the-art* SH approach to this broader dataset.
- The introduction of a more efficient CNN prediction model that maintains near-perfect accuracy while significantly reducing computation time.
- A comprehensive evaluation comparing the new CNN model to the *state-of-the-art* RNN and BRNN models in terms of accuracy, coverage, and execution time.
- A thorough performance analysis showing how both approaches handle incorrect or incomplete language derivations, demonstrating the robustness of the new CNN-based model.
- An evaluation of the speed advantages gained by running these models on GPUs, with the CNN implementation achieving up to 26.8 times faster prediction speeds compared to the BRNN and RNN models on the same hardware.

The implementation, benchmark datasets, and results are available in the replication package of this work [4].

This paper is organized in the following manner: Section II outlines the approach's design. Section III details the experimental framework, while Section IV provides and analyses the findings. Section V reviews work related to this study, and Section VI offers a conclusion that summarises the key outcomes and contributions, and considers directions for future investigations in this field.

II. Approach

The approach outlined in this paper is dedicated to addressing the challenges presented in this field. The primary goal is to expand the dataset following the strategy introduced in the *state-of-the-art* approach [3] and introduce a novel, CNN-based, approach to enhance the evaluation speed.

The devised strategy continues to focus on the development of CNNs models capable of statistically inferring the optimal behaviour of brute-force (BF) models. To achieve this objective, an oracle of syntax highlighting (SH) solutions is created by employing the language-specific BF resolver.

The subsequent sections provide a comprehensive specification of both the BF and CNN models, along with the rationale behind their design.

A. Creation of Syntax Highlighting Oracles

In this study, language highlighting oracles are generated for three new programming languages: *JavaScript*, *C++*, and *C#*, extending from the original dataset consisting of *Java*, *Kotlin*, and *Python*. This is achieved through the application of the BF model, which is tailored for each specific language to compute precise SH assignments for individual source code files.

The BF model involves two key components. Initially, it utilises the existing lexer of the target programming language to tokenize the source code, resulting in a stream of tokens. Afterwards, these tokens are structured into an AST using the language's parser.

Subsequently, a tree-walking process harnesses the structural information within the AST to assign each token to its corresponding SH class based on the grammatical context in which it is employed. The primary objective is to ensure accurate association between each token and its designated SH class, thereby achieving the highest possible highlighting accuracy for a given language [3].

It is essential to note that the BF model requires the implementation of a walker which typically consists of a small number of detection rules, as outlined in the replication package [4]. This approach ensures the deterministic creation of BF highlighters and demands only a fundamental understanding of the language's grammar, as the core lexer and parser tools specific to each language are reused. This method represents a significant departure from the conventional and error-prone processes of defining complex systems of regular expressions.

The BF model plays a pivotal role in generating the *oracle*, which is a compilation of the source code files for the target programming language, along with their corresponding SH assignments. To accomplish this, each source code file undergoes two primary steps.

Tokenisation: The file is processed through the language's lexer, which dissects the code into tokens. Each token is then transformed into an Extended Token Annotation (ETA) entity, represented as $ETA = \{i_s, i_e, t, tr\}$. In this representation: 1) i_s and i_e denote the token's character start and end indexes within the file. 2) t represents the precise text referenced by the token. 3) tr signifies the token's unique *Token Rule* encoded as a natural number (or ID) consistently assigned by the language's lexer. This ID signifies the token type, such as keywords, operators, or literals. For example, `String lang = "Java"`; would result in a set of ETAs: `{0, 5, String, 102 }`, `{7, 10, lang, 102}`, `{12, 12, =, 73}`, `{14, 20, "Java", 55}`, and `{21, 21, ;, 63}`, where *tr* values represent token types.

AST Construction: The language's parser is then employed to structure these tokens into an AST based on the language's grammar specifications.

The BF resolver function for a given language L can be represented as follows:

$$bf_L : \{c\}, le_L, l_L, p_L, ws_L \rightarrow \{HETA\}$$

Where: 1) c is the character set of the input file 2) le_L signifies the lexer encoder, which transforms character sets and lexer information into ETAs. 3) l_L is the lexer of L 4) p_L represents the parser responsible for generating the language-specific AST. 5) ws_L denotes the walking strategy that maps ETAs to *Highlighted Extended Token Annotation (HETA)* objects. These HETA objects extend ETAs by including a *Highlighting Class* (hc) corresponding to the grammatical SH class to which the token belongs. Tokens that do not form part of any grammatical construction are assigned to the unique hc *ANY*, representing unhighlighted text.

This methodology enables the generalisation of SH patterns based on the sequence of language features and abstracts away the noise introduced by specific token text features, facilitating the parsing of code.

B. Coverage Tasks

This approach considers the definition of syntax highlighting coverage classes in line with previous work [3], which assumes that effective coverage tasks can be developed by aligning them with the syntactic and semantic features emphasised by modern Integrated Development Environments (IDEs), such as INTELLIJ IDEA, PYCHARM, and VISUAL STUDIO, ensuring both practical relevance and consistency across different programming environments.

Each *Coverage Task* (*T*) is therefore constructed by combining one or more of the following language feature groups. Each feature represents a unique *hc* (*Highlighting Class*; see Section II-A), or visually, a distinct color.

`Lexical`: This group includes token classes that are lexically identifiable, meaning that for a given token, only its *tr* value is required to determine its classification:

- `KEYWORD`, referring to strong keywords only, as soft keywords may also be used as user-defined identifiers in certain language contexts. This class also includes tokens of primitive types, e.g., `int`, `float` according to the language;
- `LITERAL`, encompassing any literal value in the language, e.g., numbers (integers, floating-point, binary, hexadecimal), boolean values, and null constants;
- `CHAR_STRING_LITERAL`, covering any user-defined string or character literals, including those within string interpolation sequences;
- `COMMENT`.

For this group, all classes are assigned uniformly across the selected programming languages.

`Identifier:` This group includes classes for special types of identifiers:

- `TYPE_IDENTIFIER`, matching all identifier tokens within the language's productions that represent a type entity;
- `FUNCTION_IDENTIFIER`, including all identifiers used in function or method calls;
- `FIELD_IDENTIFIER`, referring to identifiers that grammars recognize as references to an attribute of an object or entity. These are usually preceded by an entity navigation operator, e.g., in JAVA's `Object o = a.b.c().d;`, `b` and `d` are `FIELD_IDENTIFIERS`, while `c` might be considered a `FUNCTION_IDENTIFIER`.

`Declarator:` This group includes classes for classifying token identifiers that name new top-level features of programs:

- `CLASS_DECLARATOR`, referencing identifiers bound to any newly defined declaration of a class, object, enumeration, data class, structure, etc.;
- `FUNCTION_DECLARATOR`, for identifiers bound to newly defined methods or functions;
- `VARIABLE_DECLARATOR`, for identifiers bound to newly defined variables. Note that this class is excluded from PYTHON experiments due to the intrinsic ambiguity in value-to-identifier assignments.

`Annotation:` This group includes base annotation components:

- `ANNOTATION_DECLARATOR`, referencing the token identifiers and prefixed symbols, such as `@`, of an annotation, as is common practice across the three selected languages.

Finally, the *hc* `ANY` gathers all tokens not belonging to any of the categories mentioned above.

Based on these *hc* groups, four coverage tasks are defined to evaluate the flexibility of the RNN approach in complying with various SH coverage requirements. These four *Coverage Tasks* are defined as follows:

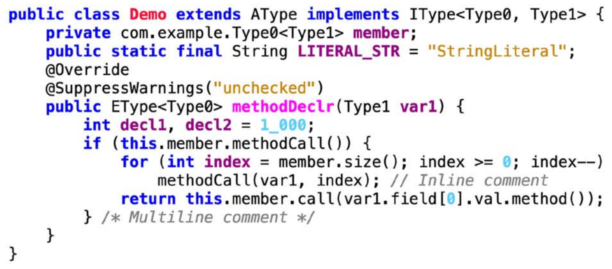
- T1: {`ANY`}, `Lexical`, `Declarator`;
- T2: {`ANY`}, `Lexical`, `Identifier`;
- T3: {`ANY`}, `Lexical`, `Declarator`, `Identifier`;
- T4: {`ANY`}, `Lexical`, `Declarator`, `Identifier`, `Annotation`.

Further details about the language groups and their detection strategy across the three additional investigated languages in this work are available in the replication package [4]. Please refer to Fig. 5 for a visual representation of the four tasks, illustrating the corresponding syntax highlighting in a Java snippet.

C. Deep Learning for Syntax Highlighting

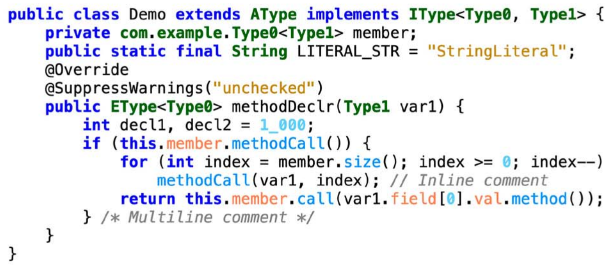
The *state of the art* approach to efficient syntax highlighting (SH) involved the use of RNNs to map sequences of token rules $\{tr\}$ to sequences of SH classes $\{hc\}$, mirroring the process performed by BF resolvers. This approach reduced the SH task to a statistical inference on the expected grammatical structure of the input token sequence.

The rationale for employing Neural Networks (NNs) in this task stemmed from the structured nature of programming language files. Programming languages exhibit the following characteristics: 1) They represent character sequences selected from



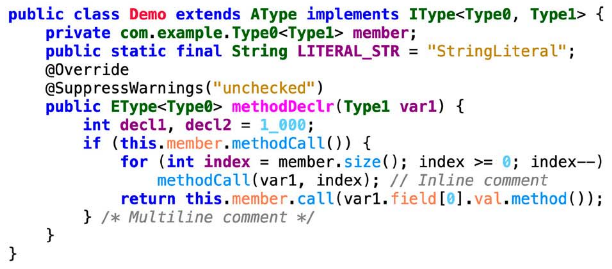
```java
public class Demo extends AType implements IType<Type0, Type1> {
    private com.example.Type0<Type1> member;
    public static final String LITERAL_STR = "StringLiteral";
    @Override
    @SuppressWarnings("unchecked")
    public EType<Type0> methodDeclr(Type1 var1) {
        int decl1, decl2 = 1_000;
        if (this.member.methodCall()) {
            for (int index = member.size(); index >= 0; index--)
                methodCall(var1, index); // Inline comment
            return this.member.call(var1.field[0].val.method());
        } /* Multiline comment */
    }
}
```

Fig. 1: Example Task Coverage *T1*



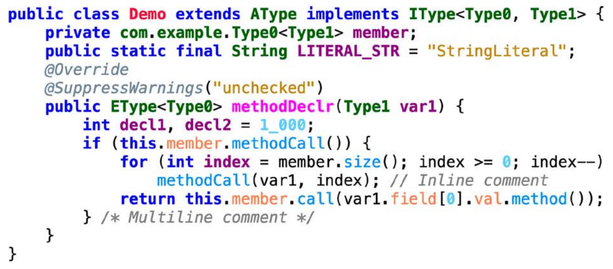
```java
public class Demo extends AType implements IType<Type0, Type1> {
    private com.example.Type0<Type1> member;
    public static final String LITERAL_STR = "StringLiteral";
    @Override
    @SuppressWarnings("unchecked")
    public EType<Type0> methodDeclr(Type1 var1) {
        int decl1, decl2 = 1_000;
        if (this.member.methodCall()) {
            for (int index = member.size(); index >= 0; index--)
                methodCall(var1, index); // Inline comment
            return this.member.call(var1.field[0].val.method());
        } /* Multiline comment */
    }
}
```

Fig. 2: Example Task Coverage *T2*



```java
public class Demo extends AType implements IType<Type0, Type1> {
    private com.example.Type0<Type1> member;
    public static final String LITERAL_STR = "StringLiteral";
    @Override
    @SuppressWarnings("unchecked")
    public EType<Type0> methodDeclr(Type1 var1) {
        int decl1, decl2 = 1_000;
        if (this.member.methodCall()) {
            for (int index = member.size(); index >= 0; index--)
                methodCall(var1, index); // Inline comment
            return this.member.call(var1.field[0].val.method());
        } /* Multiline comment */
    }
}
```

Fig. 3: Example Task Coverage *T3*



```java
public class Demo extends AType implements IType<Type0, Type1> {
    private com.example.Type0<Type1> member;
    public static final String LITERAL_STR = "StringLiteral";
    @Override
    @SuppressWarnings("unchecked")
    public EType<Type0> methodDeclr(Type1 var1) {
        int decl1, decl2 = 1_000;
        if (this.member.methodCall()) {
            for (int index = member.size(); index >= 0; index--)
                methodCall(var1, index); // Inline comment
            return this.member.call(var1.field[0].val.method());
        } /* Multiline comment */
    }
}
```

Fig. 4: Example Task Coverage *T4*

Fig. 5. A visual representation of grammatical syntax highlighting for each defined Coverage Task.

a finite set of terminal symbols $\{tr\}$. 2) These sequences adhere to an underlying formal grammar, which imposes a pure ordering function. SH represents a grammar for which there always exists a correct language derivation when a valid derivation of the original grammar exists. In other words, the SH grammar, denoted as hg, parses sub-productions s_{hg} of the original grammar g sequentially. These sub-productions are sufficient to discriminate a *tr* subsequence for a target highlighting construction. Alternatively, they map every token not consumable by any s_{hg} to a terminal symbol. With the Neural Network (NN) approach to SH, the burden of producing the SH grammar is shifted from developers to Neural Networks (NNs), which infer it from the observed behaviour of a BF model. The SH task is effectively transformed into a *sequence-to-sequence* translation task [5], converting sequences of $\{tr\}$ into sequences of $\{hc\}$. To address this problem reduction, the previous approach employed RNNs [6] for learning the bindings between SH sequences. RNNs are well-suited for sequence translation, iterating through the input sequence to produce a translation unit while retaining information to aid predictions for subsequent inputs. For grammars producing sequence where binding an *hc* to a *tr* may require looking ahead over an arbitrary number of tokens, the approach turned to BRNNs [7]. These BRNNs behave like traditional RNN but infer translations from both forward and reverse sequences, addressing this specific requirement. The NN model was designed to generate a categorical probability distribution for each *tr* over the available *hc* set. These distributions were normalised using a *softmax* function, allowing the selection of the *hc* with the highest probability for each *tr*. In the context of SH, an RNN model M can be represented as a function: $M : \{tr\} \rightarrow \{hc\}$.

This work aims to leverage the same motivations and strategies from the previous approach but introduce a novel support for CNN models for syntax highlighting. The motivation behind this shift lies in the previous approach's near-perfect SH accuracy and the inherent parallelisation design of CNN models, which can significantly reduce computation time. This transition to CNNs aims to retain the accuracy achieved by RNNs while enabling faster computations, particularly essential when dealing with large datasets and multiple programming languages. CNNs are well-suited for tasks involving structured data, such as syntax highlighting, due to their ability to capture local patterns efficiently. While more recent techniques have evolved, CNNs offer a lightweight yet robust solution for this specific application, ensuring stable performance across various programming languages and coverage settings.

D. CNN Model

Convolutional Neural Network (CNN) models have demonstrated their effectiveness in sequence-to-sequence learning tasks, surpassing the capabilities of traditional recurrent models [8], [9]. One of the key advantages of CNNs lies in their inherent ability to enable fully parallelised training, optimising the utilisation of *GPU* hardware. This parallel processing not only enhances training speed but also boosts performance in both training and inference stages [8], [9]. Additionally, CNNs serve as adept feature extractors, capable of discerning meaningful representations even from limited training data. This feature extraction ability not only offers regularisation benefits in tasks with small datasets but also ensures an expanded receptive field on the input. This receptive field grows with the number of layers, enabling the model to effectively capture dependencies across input segments. The benefits of CNNs are leveraged to improve the previously established approach and mirror the functionalities of the RNNs.

TABLE I
HYPERPARAMETERS

Parameter	Values	Selected
Kernel size	3, 5 and 7	5
Embedding dimension	16, 32, 64 and 128	32
Hidden dimension	16, 32, 64 and 128	32
Dropout	0.3, 0.4 and 0.5	0.3
Stride	1, 3 and 5	1
Padding	$\frac{kernel\_size}{2}$	1
Layers	0, 1, 2 and 3	0

A streamlined CNN model, inspired by the model proposed by Ngoc et al. [9], is introduced for its close alignment with the requirements of the SH task. The method operates on the tokenised sequence of the syntax structure, denoted as X_{in}, where X_{in} represents the number of input channels. Notably, each element $x \in X$ falls within the range of [0, 256], with each value representing a keyword requiring highlighting.

Our Convolutional Neural Network (CNN) implementation consists of an embedding layer ($\text{Emb} : \mathbb{N}^{vocab\_size} \to \mathbb{R}^{32}$)) followed by two convolutional layers ($C_i : \mathbb{R}^{32} \to \mathbb{R}^{32}$) activated by ReLU ($\sigma$), each processing the input in a different direction. Dropout regularisation ($\delta(p = 0.3)$) is applied to these first two convolutional layers to prevent overfitting. The concatenated (denoted by $\oplus$) outputs from these layers are passed to a third convolutional layer ($C_3 : \mathbb{R}^{2*32} \to \mathbb{R}^{256}$). Finally, the features are fed into a fully connected feedforward layer ($FC : \mathbb{R}^{256} \to \mathbb{N}^{12}$) that classifies the output into the respective output classes - namely, the highlighting classes *hc*. The model can be formalised as follows:

$$\begin{aligned} emb &= Emb(x) & Emb &: \mathbb{N}^{vocab\_size} \to \mathbb{R}^{32} \\ C_{concat} &= \delta(\sigma(C_1(emb)) \oplus \sigma(C_2(emb))) & C_i &: \mathbb{R}^{32} \to \mathbb{R}^{32} \\ C_{out} &= \sigma(C_3(C_{concat})) & C_3 &: \mathbb{R}^{2*32} \to \mathbb{R}^{256} \\ y &= FC(C_{out}) & FC &: \mathbb{R}^{256} \to \mathbb{N}^{12} \end{aligned}$$

In case of additional convolutional layers $C_l : \mathbb{R}^{2*32} \to \mathbb{R}^{2*32}$, indicated by *layers*, are added before the third convolutional layer to increase the depth of the network. Each of these is activated by ReLU and followed by dropout regularisation. All of the layers use the following hyperparameters (where applicable): kernel size = 5, stride = 1, padding = kernel size/2, layers = 0. Additional configurations of the models were evaluated with the hyperparameters shown in Table I, while all of the configurations converged; larger dimensions and more depths (*layers*) yielded no improvements and smaller ones reduced the performance. Therefore, the aformentioned configuration yields the best performance while also producing the smallest model.

III. EXPERIMENTS

This paper addresses seven critical research questions that collectively provide a comprehensive understanding of the performance, efficiency, and generalisation capabilities of the *state-of-the-art sh* models, with a particular emphasis on the impact of the proposed CNN model and *GPU* accelerations. The first three research questions examine the generalisation of the *state-of-the-art* RNN and BRNN models when applied to an extended dataset, considering the accuracy, prediction delays, and accuracy in handling code snippets. Following this, the same set of questions is evaluated in the context of the proposed CNN models, thus verifying that the proposed models too can provide near-perfect accuracy like *state-of-the-art* models, but with smaller prediction delays. Finally, the last research question focuses on evaluating the speed-ups that both the *state-of-the-art* RNN and the proposed CNN models can experience when evaluated on *GPU*s.

RQ1 *To what extent does the original NN-based approach maintain its near-perfect accuracy when applied to a broader set of mainstream programming languages and various levels of grammatical coverage?*

This questions evaluates to what extent the *state-of-the-art* approach for *On-the-Fly* SH can maintain its near-perfect accuracy score, for any level of coverage, to a new set of programming languages.

RQ2 *How does the prediction speed of the three SH approaches continue to compare on the mainstream programming languages?*

With the speed of evaluation being an important factor of *On-the-Fly* SH, this question provides an overview of the time delays requested by each SH resolver.

RQ3 *Compared to the regular expression (regex) and BF approaches, to what extent can the original RNN based approach continue to produce near-perfect SH solutions for incorrect or incomplete language derivations, on a new set of programming languages?*

With online SH requiring the highlighting of incorrect languages derivations (such as snippets, diffs, or different language versions), this questions investigates how the originally proposed NN-based approach can continue to deliver its accuracy gains on a new set of programming languages.

RQ4: *How does the proposed CNN model compare to the original RNN resolvers in terms of SH accuracy?*

With the original approach delivering near-perfect SH accuracy, this question aims at evaluating whether the proposed CNN model can stack up to these similar figures.

RQ5 *How does the prediction delays of the proposed CNN model compare to the original NN resolvers?*

This question evaluates whether the proposed CNN model can indeed provide shorter evaluation delays over the baseline NN solution.

RQ6 *How accurately can the proposed CNN models provide SH for incorrect or incomplete language derivations?*

This question reports on the suitability of the proposed approach in providing SH for incorrect language derivations, following similar motivations of RQ4.

RQ7 *How does the utilisation of* GPUs *impact the prediction delays of the* state-of-the-art *RNN and BRNN, and the proposed CNN models?*

TABLE II
METRICS FOR *JAVA*, *KOTLIN*, *PYTHON*, *C++*, *C#*, AND *JAVASCRIPT* NORMALISED *SH* ORACLES

Metric	Java					Kotlin					Python				
	Mean	**SD**	**Min**	**Median**	**Max**	**Mean**	**SD**	**Min**	**Median**	**Max**	**Mean**	**SD**	**Min**	**Median**	**Max**
Chars	6,239	11,575	0	2,932	504,059	2,455	4,385	80	1,490	176,176	7,391	34,325	0	3,398	3,987,090
Whitespaces	1,207	2,417	0	529	72,702	575	1,276	6	282	47,495	1,999	12,941	0	829	1,465,856
Lines	190	332	1	94	14,628	70	121	5	43	4,734	208	873	1	104	89,373
Tokens	882	1,745	1	371	45,229	737	1,559	23	327	72,484	1,161	4,997	1	525	448,562
Metric	**C++**					**C#**					**JavaScript**				
Chars	27,095	392,923	0	2,710	23,944,620	7,016	28,002	0	2,325	2,811,507	14,774	132,637	0	1,843	8,937,963
Whitespaces	8,793	140,098	0	436	7,160,063	2,199	7,282	0	578	407,133	3,437	34,329	0	428	2,933,810
Lines	427	3,762	1	94	222,894	179	442	1	68	13,293	350	3,125	1	62	277,838
Tokens	750	4,082	1	186	222,265	1,253	4,894	1	405	326,685	3,576	33,011	1	405	2,191,766

Given the time-sensitive nature of SH in online environments, this research question investigates the potential for speed-up gains by harnessing the computational power of *GPU*s. Such question is motivated by the renowned ability of *GPU*s to accelerate the execution of deep learning models such as the *state-of-the-art* RNN and BRNN, and the proposed CNN models.

A. Data, Entities and Metrics

This section outlines the entities and metrics relevant to the experiments, including the models under investigation, training procedures, cross-validation setup, coverage tasks, accuracy evaluation, benchmarks, and snippet evaluation.

Coverage Tasks. In the experiments, SH models are validated based on their ability to highlight characters in language derivation, as per three coverage tasks (*T1*, *T2*, *T3*, *T4*). These tasks follow the validation strategy employed in the *state-of-the-art* approach. Each Coverage Task (*T*) is constructed by grouping various language feature categories, each representing a unique *hc*. These categories include lexically identifiable token classes, special types of identifiers, and classes for the classification of token identifiers [3]. These coverage tasks are designed to evaluate the models' adaptability to different SH coverage requirements, ensuring that they can identify and highlight specific language features. The specific criteria for each coverage task involve detecting different feature groups, including identifiers, literals, and annotations, in the code. More details about the language feature groups and their detection strategy can be found in the replication package.

Extended Dataset. The dataset extension process, involving the inclusion of *JavaScript*, *C++*, and *C#*, closely mirrors the methodology employed in Palma et. al [3], with the primary difference being the development of new highlighting AST walkers for these three additional programming languages. Just as in the prior approach, the process starts with data mining. GitHub's public repositories are accessed through the GitHub API, and files matching the respective language extensions are collected. The collected files must be those for which the BF strategy can derive an AST. For each programming language, the data collection continues until a sample size of $20,000$ unique files is reached, with uniqueness here being determined at a token-id sequence level [3]. This sample size allows for comprehensive accuracy and performance testing. The collected datasets for each language include statistics on various aspects, such as the number of characters, whitespace, lines of code, and tokens, as reported in Table II.

The extended dataset adheres to the structure of the original dataset also with regards to the coverage tasks. In fact, each of the language datasets is duplicated four times, with each duplication configuring the SH targets to correspond to the specific colours associated with one of the four coverage tasks. This adaptation of highlighting targets is accomplished using the *Task Adapter* concept for which targets for *T1*, *T2*, and *T3* are reductions of the targets or *T4* [3].

Cross-Validation Setup. All accuracy experiments are validated using a three-fold cross-validation setup. The dataset for each language's coverage task is partitioned into three distinct folds, with each fold consisting of a training, testing and validation dataset. These splits entail a 33%-66% division into testing and training sets, with a 10% subset of the training data reserved for validation. Additionally, following the steps of [3], for each fold, the test subset is used as source set for the generation of the 5000 incorrect derivations validation dataset.

Incorrect Derivations. To evaluate the accuracy of SH for incomplete (invalid) language derivations, the approach focuses on generating invalid language derivations from a set of valid sampled files. This shift in focus is due to the infeasibility of generating correct SH for incorrect files using the BF method. For this purpose, test files are sampled line-wise to create files of snippet size. The snippet lengths are determined based on the language's mean, standard deviation, minimum, and maximum snippet line numbers, obtained from StackExchange's Data Explorer. For each language, at time of testing these amounted to (mean, standard deviation, minimum, maximum): 23.23, 15.00, 1, and 1,157 for *JavaScript*; 17.00, 28.71, 1, and 1,234 for *C++*; and 17.00, 26.89, 1, and 1,218 for *C#*. Snippets are drawn randomly from the test files, and the process generates HETAs for these snippets.

Models Under Investigation. The experiment setup aim at investigating the SH performances of this set of SH models:

1) **Brute Force:** BF models, based on *ANTLR4* [10] carried over from prior work for *Java*, *Python*, and *Kotlin*, and introduced anew in this paper for *JavaScript*, *C++* and *C#*.
2) **State-of-Practice:** Compared to previous efforts in this space which considered only *Pygments* [11], this work

evaluates the performances of both *Pygments* and another popular tool in this space: *TreeSitter* [12]. For *Pygments*, all experiment setups report updated metrics for the latest version at the time of writing *v2.13*. For *TreeSitter*, the latest version compatible with the latest version available of each official language grammar is used. Further details on versions and compatibility are available in the replication package [4].

3) **State-of-the-Art Models:** These include RNN models with different hidden unit sizes (RNN16, RNN32, RNN64) and the all the bidirectional variants (BRNN16, BRNN32, BRNN64).
4) **Proposed Models:** These include the CNN models introduced by this work (CNN32, CNN64 and CNN128).

Training Procedures. All deep learning models presented in this work (RNN, BRNN and CNN) are trained using the same training configuration presented in *Palma* et al. [3], which instructs about the choice of optimiser, learning rate, batch size, and epoch count. Hence each model is trained sequentially on each training sample, with cross-entropy loss and Adam optimiser. The training session for any SH RNN, language and coverage, was accordingly set to train for two epochs with a learning rate of 10^{-3}, and for a subsequent two epochs with a learning rate of 10^{-3}.

Accuracy Metric. Adhering to the methodology of the prior approach, this investigation focuses on ascertaining the highlighters' capacity to associate each character in the input text with the correct SH class, for each coverage specification. This approach also serves to reconcile potential discrepancies in tokenisation between the BF and Regex strategies for a given input file.

Benchmarks. The prediction speed measurement during the experiments involves evaluating the time delays for each SH resolver. This metric quantifies the absolute time in nanoseconds required to predict the SH of an input file. The benchmarking encompasses various SH methods, and adheres with the measurement techniques of *state-of-the-art* [3]. For each model the following time delays are evaluated:

- **Brute-Force:** The time taken to natively lex and parse the input file and execute a SH walk on the acquired AST.
- **State-of-Practice:** For *Pygments*, this is the time take for computing the output vector of SH classes, given the source text of the file. It excludes the time consumed for formatting the output according to any specific specification to emphasise the intrinsic time complexity of the underlying SH strategy. For *TreeSitter*, this only considers the time to process and extract the highlighting from the text of the file from a fully initialised language highlighter object. Similarly to *Pygments* and for the same goals, it excludes the process of extraction of these highlighting outputs into a formatter output object.
- **Neural Networks:** This measurement comprises two components. First, the time required for the lexer inherited from *ANTLR4* (the same lexer employed by the BF approach) to tokenize the input file, resulting in a sequence of token rules. Subsequently, it factors in the time for the NN model to create the input tensor and predict the complete output vector of SH classes.

B. Evaluating the Generalisation of the (B)RNN Approach

This section addresses the experimental setups for investigation into the generalisation of the *state-of-the-art* RNN approach for SH. This interests RQ1, RQ2 and RQ3. The primary objective is to assess the models' capacity to deliver similar levels of performance in a broader range of programming languages. The performance metrics under scrutiny encompass accuracy, coverage, and the execution time of the models.

To accomplish this, an extended dataset was curated to include three additional mainstream programming languages: *JavaScript*, *C++*, and *C#*. This expansion followed the strategy introduced in the previous approach, focusing on the creation of language-specific oracle models based on the languages' original lexers and parsers, relying on the *ANTLR4* [10] grammars for each language.

Following the original training configuration, RNN16, RNN32, BRNN16 and BRNN32 models were trained for each coverage task for every new language. In particular, these were trained on the training set of each task fold. Furthermore, trainings were performed on the same hardware configuration originally used. In the evaluation phase, the performance of the resulting RNN models, the *state-of-practice*, and BF resolvers were examined across several dimensions. For assessing accuracy, each RNN model was tested on the dedicated test dataset corresponding to the fold on which it was trained. Similarly, the *state-of-practice* resolvers underwent accuracy testing on the same fold test datasets. Notably, the BF method, which is known for producing perfect predictions, was not included in accuracy testing. For the evaluation of incorrect derivations, all resolvers, including BF, were subjected to testing on the incomplete test dataset for each fold. In this context, the resolution process remained consistent across all methods. Speed benchmarks were recorded differently, with timing measurements conducted for each resolver on the entire dataset consisting of $20,000$ files for each programming language and averaged across 30 reruns per file. Such conditions, together with the details discussed earlier, ensure that the models are trained and evaluated under the same circumstances as those applied in the evaluation of the *state-of-the-art* approach [3].

C. Evaluating the CNN Models

This set of experiments aims to assess the effectiveness of the newly proposed CNN resolvers in achieving SH accuracy on par with or exceeding the RNN solutions, while achieving faster prediction times. The experiments are designed to address the research questions RQ4, RQ5, and RQ6.

The evaluation follows the same validation processed carried out by *state-of-the-art* on the RNN approach, utilising the extended dataset previously described in Section III-B, which covers languages such as *Java*, *JavaScript*, *Kotlin*, *Python*, *C++*, and *C#*. The proposed CNN models, specifically *CNN64* and *CNN128*, are trained using the same training configuration originally developed for the RNN and BRNN models.

In the preliminary analysis, hyperparameters were investigated using the *Java* validation dataset as a reference, with the objective of determining optimal settings for the CNN model. This exploration revealed that smaller kernel sizes, specifically values of 3, 5, and 7, produced the most effective results for the defined objectives. The choice of smaller kernel sizes was informed by the immediate relationships between elements in the sequence, which are characteristic of the short-term connections prevalent in the SH task. Unlike translation tasks that heavily rely on long-term dependencies between words, the nature of syntax highlighting called for a more nuanced approach, favouring smaller kernel sizes. The exploration also involved varying the number of layers in the CNN stack, from 1 to 4 layers, with fewer layers emerging as the superior choice, likely due to the unique demands of the task. Consistency was maintained by using the same hidden and embedding dimensions employed in the previous RNN models, with dimensions ranging from 2^4 to 2^8. To simplify the exploration, different dropout values were tested within a range of 0.1 to 0.5. Initially, the embedding size was fixed, and then systematically increased. Subsequent investigations involved expanding hidden dimensions and other parameters. Three model configurations, where performance converged, were selected for thorough evaluation. Hence this set of experiments evaluates the following models: CNN32 features a single layer with 32 hidden units and a 32-layer embedding, CNN64 features one layer with 64 hidden units and a 64-layer embedding, and CNN128 features a single layer with 128 hidden units and a 128-layer embedding.

The evaluation of the newly introduced CNN models follows a comprehensive approach. These CNN models were trained on all six programming languages within the extended dataset, and for each of the four defined coverage tasks, similar to the RNN models. Accuracy testing was conducted on the complete and incorrect files, mirroring the evaluation process of the RNN models. Additionally, the benchmarking of CNN models was carried out in the same manner as the RNN models. It is important to note that the CNN models underwent training and benchmarking on the same hardware configuration used for the RNN models. This consistency in testing procedures and hardware configuration facilitates a direct and meaningful comparison between the *state-of-the-art* RNN models and the novel CNN models.

D. Evaluating GPU Speed-Ups

This experimental setup interest RQ7, and focus on the evaluation of execution speed improvements attained by employing *GPU*s for SH prediction. It includes a comparison between the *state-of-the-art* RNN models (RNN16, RNN32, BRNN16 and BRNN32) and the proposed CNN models (CNN32, CNN64, CNN128) concerning prediction speed when using a *GPU*.

For each of the six languages in the extended dataset (*Java*, *JavaScript*, *Kotlin*, *Python*, *C++*, and *C#*), SH predictions were conducted on each of the $20,000$ files contained in the respective language dataset. As the speed evaluations conducted in [3], these predictions were repeated 30 times for robustness and consistent evaluation.

The time delays for SH prediction take into account both lexing and model prediction, mirroring the parameters used in the previous *state-of-the-art* approach. However, in this set of experiments, the crucial difference lies in the execution of model evaluation on a GPU. Further implementation details and information are provided in the associated replication package.

E. Execution Setup

All RNN and CNN models are trained on a machine equipped with an AMD EPYC 7702 64-Core CPU clocked at 2.00GHz, 64GB of RAM, and a single Nvidia Tesla T4 GPU with 16GB of memory. The same machine is utilised for GPU benchmarking experiments. Instead, all performance testing for all of the compared approaches was carried out on the same machine with an 8-Core Intel Xeon(R) Gold 6126 CPU clocked at 2.60GHz with 62 GB of RAM.

F. Threats to Validity

The adoption of ANTLR4 as a unified framework for defining and evaluating the BF model in the context of real-time SH presents a well-rounded choice. However, the existence of alternative parsing tools, some of which might be tailored to specific programming languages, could potentially influence the efficacy of BF resolvers. The selection of such tools should be aligned with the practical demands of online SH, as outlined in Section I.

A notable challenge is the reliance on synthetically generated, incomplete or incorrect language constructs to meet requirements RQ3 and RQ6. This synthetic approach, while practical, lacks the direct correlation with real-world user-generated code snippets, necessitating a cautious interpretation of results. Despite this, the synthetic dataset serves to indirectly verify the model's capacity to deduce likely missing contexts, although it introduces a degree of variability inherent to its manual creation process, thus underlining the reliance on statistical approximations.

This study's comparison with PYGMENTS, which supports a vast array more than 500 languages, adds significant value. Nevertheless, the limitation stems from comparing only a subset of languages (JAVA, KOTLIN, PYTHON, *C++*, *C#*, and *JavaScript*), suggesting broader applicability through language-specific BF training. A more comprehensive evaluation across all languages supported by REGEX-based alternatives would enhance the understanding of the proposed approach's abilities.

Moreover, the predictive delay benchmarks, while providing an overview of tool performance, might not fully capture nuances related to specific implementation choices or factors inherent to different platforms, such as online file size constraints. Aspects like integration effectiveness, caching mechanisms,

TABLE III
MEDIAN VALUES OVER 3 FOLDS FOR THE ACCURACY. THE MAXIMUM SCORES PER TASK ARE HIGHLIGHTED

Model	**JAVA**				**KOTLIN**				**PYTHON**			
	T1	**T2**	**T3**	**T4**	**T1**	**T2**	**T3**	**T4**	**T1**	**T2**	**T3**	**T4**
REGEX	0.8668	0.7631	0.7262	0.7261	0.8005	0.6997	0.6784	0.6770	0.9364	0.8191	0.8191	0.8167
TREE-SITTER	0.9271	0.7799	0.7770	0.7757	0.7104	0.6150	0.5437	0.5311	0.9091	0.9117	0.8671	0.8629
RNN(16)	0.9987	0.9716	0.9676	0.9668	1.0000	0.9627	0.9598	0.9604	1.0000	0.9560	0.9559	0.9550
RNN(32)	1.0000	0.9751	0.9710	0.9706	1.0000	0.9648	0.9640	0.9630	1.0000	0.9572	0.9571	0.9570
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(64)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(128)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Model	**C++**				**C#**				**JAVASCRIPT**			
REGEX	0.9018	0.9543	0.8743	0.8743	0.8864	0.7556	0.7317	0.7317	0.9516	0.8339	0.8077	0.8077
TREE-SITTER	0.2110	0.2912	0.1859	0.1859	0.8680	0.8238	0.7390	0.7390	0.9589	0.8542	0.8282	0.8282
RNN(16)	0.9972	1.0000	0.9893	0.9915	0.9901	0.9580	0.9468	0.9463	1.0000	0.9236	0.9186	0.9242
RNN(32)	0.9982	1.0000	0.9953	0.9953	0.9913	0.9646	0.9608	0.9603	1.0000	0.9000	0.9309	0.9299
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	0.9986	0.9979	0.9979	1.0000	1.0000	1.0000	1.0000
CNN(64)	1.0000	1.0000	1.0000	1.0000	1.0000	0.9989	0.9985	0.9985	1.0000	1.0000	1.0000	1.0000
CNN(128)	1.0000	1.0000	1.0000	1.0000	1.0000	0.9992	0.9986	0.9986	1.0000	1.0000	1.0000	1.0000

and hardware capabilities could also influence the performance of SH solutions. The efficiency of the suggested CNN approach might vary across different operational environments, such as when deployed using advanced deep learning frameworks [13] or on GPU hardware, suggesting potential avenues for further investigation and optimisation.

IV. RESULTS

Expanding upon the experimental configurations detailed in Section III, this section delves into a comprehensive analysis of the proposed approach's performance in response to the four research questions outlined. To facilitate comparisons, the "Kruskal-Wallis H" test [14] was employed in tandem with the "Vargha-Delaney $\hat{A}_{12}$" test [15] to gauge the effect size, shedding light on the magnitude of observed differences. Consequently, the ensuing discussion presents the evaluation metrics in terms of median values, a choice motivated by the tests' foundation in assessing median differences.

A. RQ1 – Generalisation: Accuracy

In response to RQ1, which examines the ability of the original NN based approach to retain its near-perfect accuracy when applied to a broader set of mainstream programming languages and various levels of grammatical coverage, significant insights were obtained. Table III summarises the results for the SH accuracy obtained by each resolver, in all combinations of task and language.

The examination of generalisation performance for SH accuracy in *C++* reveals intriguing findings. The base RNN models, RNN16 and RNN32, continue to exhibit accuracy gains akin to those observed in previous research involving *Java*, *Kotlin*, and *Python*. However, these models display slightly higher accuracy in *C++*. Their accuracy scores approach near-perfection, with exceptions noted in scenarios where correct highlighting hinges on deterministically feasible token look-ahead.

The bidirectional variants, namely BRNN16 and BRNN32, consistently deliver near-perfect accuracy scores. Surprisingly, the *state-of-the-art* resolvers demonstrate a slightly improved performance in *C++* compared to *Java*, *Kotlin*, and *Python*. However, they still remain significantly inferior to the NN based solutions. Furthermore, the distribution of accuracy scores is notably more compact and skewed towards perfect accuracy in both the RNN and BRNN models, in contrast to the *state-of-the-art* solution, which exhibits larger variance in its predictions. This pattern aligns with previous observations in *Java*, *Kotlin*, and *Python*, affirming the approach's capacity to generalize in terms of accuracy to *C++*.

The results for *C#* mirror the conclusions drawn from the *C++* analysis. The RNN models, while not performing as remarkably as in *C++*, demonstrate accuracy levels closer to those observed in *Java*, *Kotlin*, and *Python*. Therefore, the approach maintains its ability to generalize its accuracy to *C#*.

JavaScript showcases a unique scenario. Accuracy scores for the RNN models in this language are the lowest among all six languages, with median values hovering in the low 90s. However, the accuracy of the state-of-practice resolvers is consistent with what has been observed in other languages. The bidirectional networks, on the other hand, continue to deliver near-perfect accuracy. Similar to other languages, the distribution of accuracy scores is more densely concentrated towards perfect accuracy in both the RNNs and even more so in the BRNNs, compared to the *state-of-the-art* approach. It is worth noting that base RNN models exhibit a small number of results below 50% accuracy, an anomaly not observed in the other five languages.

TABLE IV
DESCRIPTIVE STATISTICS OF EXECUTION TIME (MS)

Model	Java					Kotlin					Python				
	Mean	**SD**	**Min**	**Median**	**Max**	**Mean**	**SD**	**Min**	**Median**	**Max**	**Mean**	**SD**	**Min**	**Median**	**Max**
BF	243.748	935.827	0.004	50.703	48,970.153	38.045	109.474	0.011	9.196	17,129.873	55.673	252.924	0.034	25.839	27,723.622
REGEX	0.019	0.023	0.010	0.015	3.340	0.017	0.039	0.010	0.014	4.335	0.020	0.037	0.010	0.015	4.648
TREE-SITTER	1.848	3.675	0.032	0.781	103.488	1.047	2.095	0.084	0.565	118.252	2.772	14.673	0.018	1.283	1,433.130
RNN(16)	15.754	22.715	0.347	10.795	665.778	16.416	38.120	0.612	10.839	14,940.728	81.635	297.425	0.367	44.111	28,530.784
RNN(32)	17.030	24.138	0.366	11.061	806.670	17.588	36.850	0.639	11.153	13,275.436	84.829	306.712	0.363	46.744	30,926.886
GRNN(16)	11.126	21.158	0.434	4.946	570.257	11.218	35.648	0.655	5.195	14,492.819	72.075	318.203	0.420	33.962	29,676.576
GRNN(32)	11.113	21.167	0.426	4.930	585.709	10.744	35.921	0.639	4.899	14,572.781	80.569	358.720	0.398	38.057	34,206.802
BRNN(16)	26.956	42.556	0.450	16.300	1,112.187	27.417	54.053	0.917	16.360	14,708.969	99.543	357.392	0.417	54.058	36,217.433
BRNN(32)	28.833	44.477	0.462	18.013	1,250.126	29.424	55.613	0.963	17.883	14,902.277	102.585	360.601	0.430	55.232	36,216.305
GBRNN(16)	21.501	41.888	0.544	9.303	1,152.935	19.356	49.915	0.905	8.573	15,039.211	87.273	398.539	0.465	40.864	39,490.076
GBRNN(32)	21.582	42.024	0.539	9.343	1,145.865	18.980	48.418	0.884	8.534	14,283.623	86.608	377.965	0.477	40.706	35,263.657
CNN(32)	19.225	12.019	0.673	19.746	93.318	18.431	28.639	0.753	18.613	14,284.947	77.395	268.063	0.811	47.581	28,522.633
CNN(64)	20.191	13.195	0.701	20.160	453.234	19.225	30.516	0.838	18.542	14,470.209	72.955	244.924	0.768	47.048	24,690.476
CNN(128)	21.583	14.318	0.715	20.537	223.501	20.321	30.983	0.904	18.623	14,305.525	74.613	251.771	0.883	48.048	25,858.651
GCNN(32)	0.789	0.405	0.514	0.659	40.768	1.632	28.127	0.522	0.825	14,150.149	64.155	284.315	0.539	30.500	26,769.611
GCNN(64)	0.747	0.329	0.502	0.642	39.996	1.627	28.251	0.522	0.814	14,053.773	64.444	285.610	0.525	30.616	26,863.922
GCNN(128)	0.742	0.331	0.502	0.634	40.744	1.653	29.485	0.523	0.810	14,610.805	64.237	284.855	0.497	30.535	26,817.085
Model	**C++**					**C#**					**JavaScript**				
BF	34.598	119.267	0.001	7.281	5,287.692	7.766	121.491	0.011	0.904	11,891.983	150.429	1,084.771	0.026	16.425	91,833.385
REGEX	0.034	0.296	0.010	0.015	63.802	0.023	0.045	0.010	0.016	9.708	0.030	0.152	0.010	0.015	27.014
TREE-SITTER	5.029	41.993	0.024	1.012	2,576.648	2.576	9.397	0.050	0.822	723.286	5.562	51.095	0.018	0.608	3,631.976
RNN(16)	11.038	43.606	0.237	5.280	2,589.176	19.128	52.872	0.396	11.078	3,712.954	141.235	1,177.542	0.308	25.590	82,025.300
RNN(32)	12.841	47.249	0.244	5.601	2,721.487	20.814	55.376	0.393	11.623	3,529.568	138.989	1,135.261	0.308	26.481	79,816.063
GRNN(16)	10.176	56.718	0.340	2.774	3,915.631	15.811	60.962	0.467	5.420	4,249.555	144.089	1,329.131	0.419	17.508	89,162.290
GRNN(32)	10.136	56.839	0.346	2.768	3,981.607	15.837	60.704	0.459	5.446	4,196.093	143.593	1,331.705	0.396	17.336	89,539.529
BRNN(16)	19.435	85.809	0.294	8.247	5,057.532	33.646	103.206	0.499	16.840	7,244.955	180.193	1,496.569	0.357	32.483	105,668.888
BRNN(32)	22.044	91.434	0.294	9.356	5,194.802	36.119	108.729	0.536	18.625	7,226.380	187.685	1,530.609	0.387	33.979	103,981.296
GBRNN(16)	19.533	109.116	0.399	5.087	7,759.243	30.607	119.945	0.617	10.200	8,276.269	185.262	1,720.401	0.446	22.015	119,257.848
GBRNN(32)	19.363	107.666	0.412	5.082	7,598.733	30.730	120.494	0.617	10.213	9,268.634	204.786	1,881.986	0.455	24.408	141,257.846
CNN(32)	20.206	14.235	0.536	19.720	625.615	19.568	13.550	0.676	19.910	670.424	109.029	870.394	0.694	30.075	61,679.980
CNN(64)	21.607	16.201	0.568	20.531	766.439	20.584	15.803	0.718	20.181	776.559	108.286	863.380	0.768	30.672	61,510.434
CNN(128)	22.881	21.023	0.621	20.865	1,158.937	22.309	21.045	0.773	20.704	1,766.114	114.549	941.239	0.752	32.341	80,964.860
GCNN(32)	1.004	4.861	0.487	0.649	301.837	1.003	1.390	0.515	0.728	140.178	108.637	999.694	0.529	13.740	67,784.865
GCNN(64)	1.004	4.774	0.480	0.648	311.590	0.984	1.370	0.510	0.721	138.915	96.105	877.135	0.534	12.353	58,771.929
GCNN(128)	0.986	4.720	0.451	0.633	289.385	0.996	1.391	0.519	0.728	133.209	98.953	907.218	0.500	12.678	61,571.581

Overall, the accuracy of the *Pygments* resolvers is consistent with findings from previous research on *Java*, *Kotlin*, and *Python*. However, these resolvers consistently underperform compared to all RNN and BRNN models. A slight decrease in accuracy was noted for *JavaScript*, but this issue does not affect the non-baseline bidirectional networks, which continue to exhibit near-perfect performance.

Regarding the *TreeSitter* resolvers, they are consistently outperformed by all NN-based approaches examined in this study. Furthermore, they are the least consistent performers across all languages, showing significant drops in accuracy for *C++* and *Kotlin*, which achieve a median accuracy of 18% and 53% for *T4*, respectively. Aside from the cases of *C++* and *Kotlin*, the two *state-of-practice* resolvers generally perform comparably.

B. RQ2 – Generalisation: Benchmarking

RQ2 delves into the generalisation of prediction speed for RNN and BRNN models across mainstream programming languages *Java*, *Kotlin*, *Python*, *C++*, *C#*, and *JavaScript*. The focus is on identifying whether the performance characteristics, particularly the instantaneous response time [16], observed in prior studies on *Java*, *Kotlin*, and *Python* continue to hold across the expanded set of languages.

To assess this, RNN and BRNN models, including RNN16, RNN32, BRNN16, and BRNN32, were retrained on all six languages. The models trained for *T4* were benchmarked 30 times on each of the 20*k* files in each language's dataset. These experiments were conducted on the same machine, with no *GPU* utilisation for *NN*-based resolver evaluations, ensuring consistency and comparability. The results are summarised in Table IV.

Building on prior work that classified RNN and BRNN prediction delays as within the *instantaneous* response-time category [16], this study confirms their continued efficiency in this regard. In the instantaneous category, interactions are expected to complete within 100-200 ms [16], [17], aligning with typical user actions like clicking and typing. Additionally, speed-ups of RNNs and BRNNs over BFs in *Java* and *Kotlin*, with comparable efficiency for *Python*, were previously identified and are consistent in this expanded study.

Specifically, RNN and BRNN models consistently fall within the instantaneous category, offering notable speed-ups over the BF across languages. RNN16 and RNN32 demonstrate speed-ups of 15 times for *Java*, 2 times for *Kotlin*, 3 times for *C++*, and at least 1 time faster for *JavaScript*. While performing on par with the BF for *Python*, RNN models are within the lower bound of the instantaneous category for *C#*. Similarly, BRNN16 and BRNN32 fall within the instantaneous bounds, providing speed-ups over the BF of 9 times for *Java*, performing on par for *Kotlin*, and 2 times for *C++*. However, BRNNs are 2 times slower than the BF for *Python*, at least 4 times slower for *C#*, and on par with *JavaScript*.

Overall, RNN and BRNN SH resolvers consistently operate within the instantaneous response-time category, delivering speed-ups over the BF in most cases. Exceptions exist where the BF proves to be time-wise efficient, particularly in scenarios

TABLE V
MEDIAN VALUES OVER 3 FOLDS FOR THE ACCURACY FOR SNIPPETS. THE MAXIMUM SCORES PER TASK ARE HIGHLIGHTED

Model	JAVA				KOTLIN				PYTHON			
	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
BF	0.9468	0.8420	0.7525	0.7356	1.0000	0.9827	0.9765	0.9728	1.0000	1.0000	1.0000	1.0000
REGEX	0.8856	0.7170	0.6653	0.6645	0.8447	0.6893	0.6570	0.6549	0.9401	0.8075	0.8071	0.8045
TREE-SITTER	0.9064	0.7425	0.7397	0.7376	0.8410	0.6957	0.6047	0.5882	0.9106	0.9102	0.8578	0.8525
RNN(16)	1.0000	0.9579	0.9510	0.9505	1.0000	0.9494	0.9460	0.9457	1.0000	0.9603	0.9592	0.9586
RNN(32)	1.0000	0.9632	0.9555	0.9552	1.0000	0.9528	0.9507	0.9504	1.0000	0.9616	0.9613	0.9615
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(64)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(128)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Model	C++				C#				JAVASCRIPT			
BF	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9784	0.9744	0.9744
REGEX	0.9175	0.9722	0.8779	0.8779	0.8307	0.6841	0.6561	0.6561	0.9460	0.7949	0.7653	0.7653
TREE-SITTER	0.2669	0.3852	0.2229	0.2229	0.8794	0.8049	0.6884	0.6884	0.9651	0.8244	0.7937	0.7937
RNN(16)	1.0000	1.0000	0.9918	0.9959	1.0000	0.9949	0.9891	0.9915	1.0000	0.9193	0.9161	0.9227
RNN(32)	1.0000	1.0000	0.9989	0.9957	1.0000	1.0000	1.0000	1.0000	1.0000	0.8820	0.9284	0.9264
BRNN(16)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BRNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(32)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(64)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
CNN(128)	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

with smaller file sizes. This scalability advantage of RNN and BRNN resolvers is evident when compared to the BF resolver, as illustrated in Fig. 7.

C. RQ3 – Generalisation: Accuracy on Invalid Derivations

Addressing RQ3, the investigation delved into the SH accuracy of RNN and BRNN highlighters confronted with incomplete or incorrect language derivations. Similar to RQ1, all RNN and BRNN approaches were configured to generate highlighting for all six programming languages and four coverage tasks, with results averaged across three folds. The dataset utilised for RQ3 is the generated snippet dataset, where perfect target solutions are known. The results are summarised in Table V. The findings reveal that the RNN-based approaches effectively sustain accuracy performances comparable to those achieved on language derivations where an AST is derivable. Significantly, the accuracy values observed for *Java*, *Kotlin*, and *Python* extend to *C++*, *C#*, and *JavaScript*. For RNN16, across all four tasks, the model exhibits average median accuracies of 96.49% for *Java*, 96.03% for*Kotlin*, and 96.95% for *Python*. Notably, it achieves accuracies of 99.69% for*C++*, 99.38% for *C#*, and 93.95% for *JavaScript* on the new dataset. Similarly, RNN32 demonstrates accuracy rates of 96.85% for *Java*, 93.35% for*Kotlin*, and 97.11% for *Python*, while maintaining accuracies of 99.87% for*C++*, 100% for *C#*, and 93.42% for *JavaScript*. Both RNN16 and RNN32 consistently produce near-perfect accuracies for all six languages across all tasks. Furthermore, all models significantly outperform the *state-of-practice* resolvers across all languages and tasks. While the near-perfect behaviour of the BF strategy recorded for *Python* continues for *C++* and *C#*, deviations for *JavaScript* in *T2*, *T3*, and *T4* are attributed to the snippet strategy employed in this evaluation. In terms of accuracy variance, the results illustrate that the variance is significantly greater for *Pygments*, *TreeSitter*, and BF models compared to RNN and especially BRNN models, confirming observations from previous work on the initial dataset. The near-perfect accuracy demonstrated by the proposed RNN and BRNN models on the original dataset is robustly extended to the newly introduced and more extensive dataset. This showcases the models' effectiveness in maintaining exceptional accuracy even in the face of incomplete or incorrect language derivations across various programming languages and coverage tasks.

D. RQ4 – CNN: Accuracy

RQ4 ascertains whether the CNN32, CNN64, and CNN128 models can achieve the same near-perfect levels of SH accuracy as the *state-of-the-art* RNN16, RNN32, BRNN16, and BRNN32 models. The results are derived from the per-character SH accuracy measured for each model concerning valid language derivations found in the extended SH dataset. The results are summarised in Table III. The proposed CNN models consistently deliver near-perfect SH predictions across the five programming languages of *Java*, *Kotlin*, *Python*, *C++*, and *JavaScript*. Only in the case of *C#* do the CNN models exhibit a minor deviation from this trend, with a median accuracy rate in the high 99%. Importantly, these near-perfect predictions remain consistent across all *Coverage Tasks*. Furthermore, the CNN variants consistently outperform the non-bidirectional RNN16 and RNN32 models. These base RNN models achieve comparable results in only specific tasks, such as *T1* for *Java*, *Kotlin*, *Python*, and *JavaScript*, as well as *T2* for *C++*. For each of the considered programming languages, the CNN32, CNN64, and CNN128 models consistently produce SH results

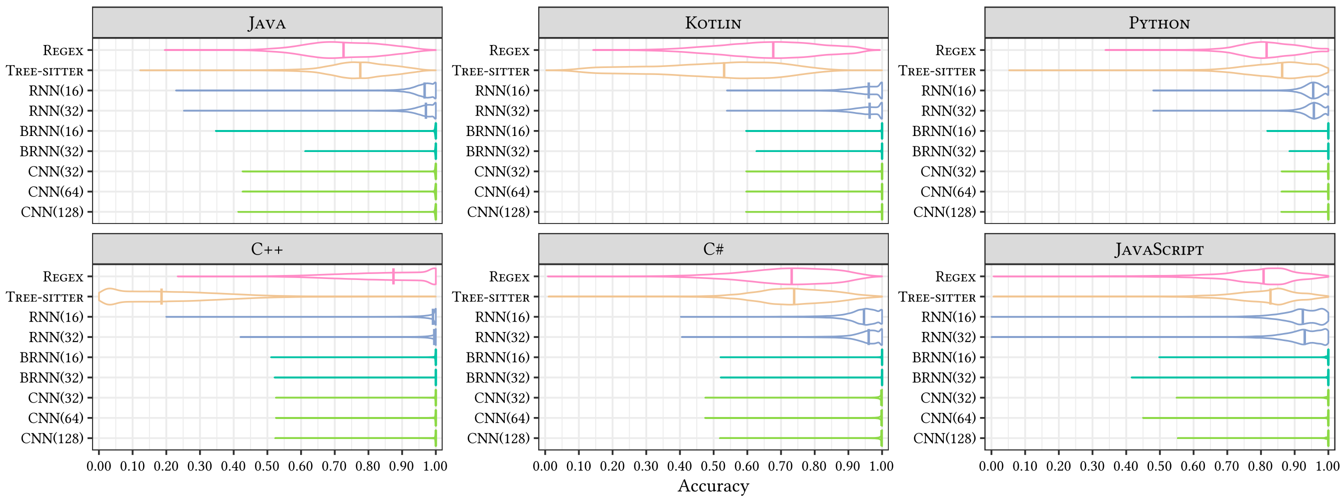


Fig. 6. Accuracy values comparison for T4.

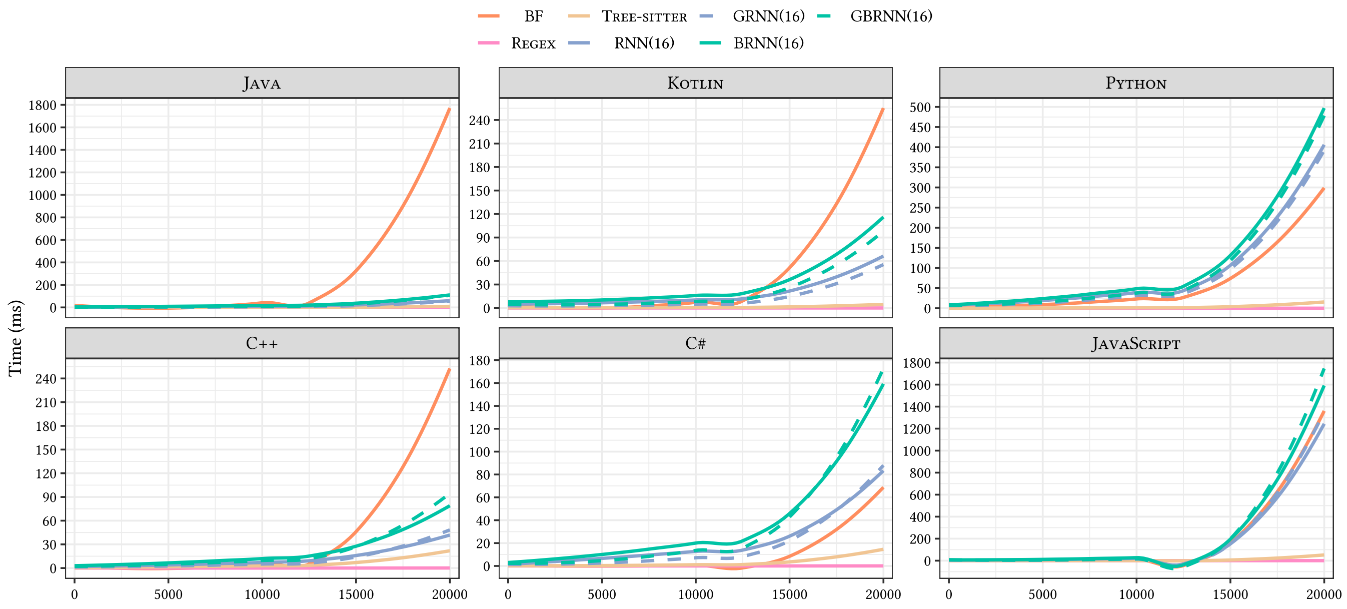


Fig. 7. Execution time (ms) values trends comparison for T4.

that are tightly clustered around perfection, with only a minor number of outliers. This phenomenon is also observed in *T4*, as depicted in Fig. 6. The three CNN variants do not introduce significant accuracy variations or trends and maintain a prediction density closely aligned with the *state-of-the-art* bidirectional models BRNN16 and BRNN32. Overall, the outcomes of RQ4 affirm that the proposed CNN solutions do not result in observable losses in SH accuracy. Instead, they demonstrate the potential to contribute on-par with the *state-of-the-art* resolvers, thus providing a robust and viable alternative for SH.

E. RQ5 – CNN: Benchmarking

RQ5 delves into the prediction speed of CNN SH models when executed on a *CPU*, specifically examining their comparison to the instantaneous [3], [16] response times observed for RNN and BRNN models. This exploration serves the dual purpose of assessing the suitability of CNNs running on *CPU* for real-time applications, adhering to Seow's response-time categorisation, where instantaneous responses complete within 100 ms to 200 ms [16]. Additionally, it seeks to identify potential speed-ups achievable through this execution approach.

Importantly, RQ7 will extend this inquiry to CNN performance on *GPU*, aligning with the proposed approach's intended usage and facilitating a comprehensive comparison against *(B)RNN* variants.

The evaluation setup mirrors that of RQ2, retraining and benchmarking CNN models (CNN32, CNN64, and CNN128) across *Java*, *Kotlin*, *Python*, *C++*, *C#*, and *JavaScript*. The

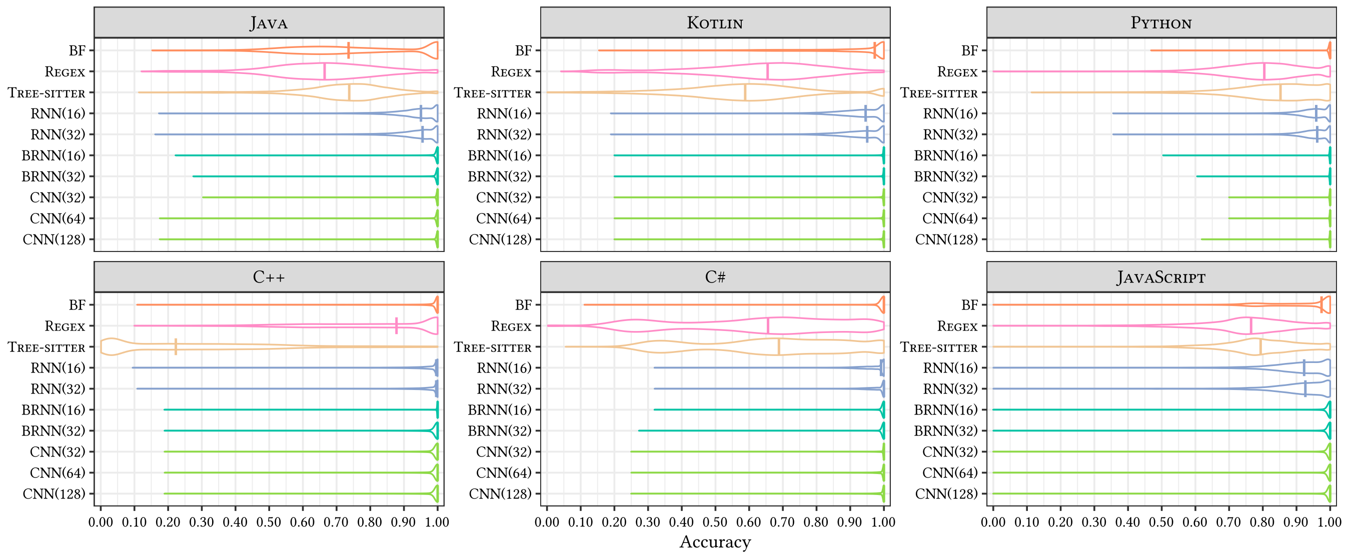


Fig. 8. Accuracy values comparison for incomplete language derivations.

benchmarking is conducted on *CPU* for *T4*. The results are summarised in Table IV.

For *Python*, CNNs emerge as the fastest NN models, surpassing RNNs by 1.1 times and BRNNs by 1.3. Similarly, in *JavaScript*, CNNs perform the best, outpacing BRNNs by 1.7 times and RNNs by 1.3. In the case of *Kotlin* and *C#*, CNNs exhibit comparable performance to RNNs while maintaining a 1.5 to 1.7 times speed advantage over BRNNs respectively. In the case of *C++*, CNNs perform on par with BRNNs, with RNNs demonstrating a 2 times speed advantage. In the case of *Java*, CNNs showcase a 1.7 times speed advantage over BRNNs, while RNNs maintain a 1.2 times speed advantage over CNNs.

Overall, when computed on *CPU*s, CNNs consistently achieve an instantaneous response time. They not only outperform BRNNs across various languages but also, in certain instances, compete favourably with the most lightweight alternatives: RNN models. These findings position CNNs as a promising choice for real-time SH applications.

F. RQ6 – CNN: Accuracy on Invalid Derivations

To comprehensively evaluate the proposed CNN models, RQ6 investigates the extent to which these models can maintain near-perfect accuracy in the face of incomplete or incorrect language derivations. Similar to RQ3, the evaluation leverages the same snippet dataset, employing an evaluation strategy consistent with the RNN models in the previous research question. Table V reports on how all CNN models achieve near-perfect accuracy, comparable to the BRNNs. As illustrated in Fig. 8, detailing the accuracy distribution for each model and language for *T4*, CNN32, CNN64, and CNN128 exhibit similar variance to BRNN32. The smaller CNN32 model provides a slightly smaller variance advantage, particularly in *Java* and *Python*. These findings affirm that the proposed CNN models can indeed offer SH of accuracy on par with the original CNN-based approach, in the case of incorrect or incomplete language derivations.

G. RQ7 – GPU Speed-Ups

RQ7 delves into the examination of prediction speed-ups for deep models in the context of SH when evaluated on a *GPU*. Following the methodology akin to RQ2 and RQ5, all models trained for T4 undergo benchmarking on the same hardware for generating SH for all 20,000 files per language, repeating this process 30 times for statistical reliability.

The summarised results in Table IV, denoted by the prefix "G" for *GPU*, reveal that *GPU* evaluation only yielded negligible improvements, with speed-ups reaching a maximum of 1.5 times faster for *Java* and *Kotlin*, followed by *C#* with 1.3, *C++* with 1.2, *Python* with 1.1, and *JavaScript* performing on par. This aligns with the non-significant architectural optimisations observed for RNNs when applied to *GPU*s. Similar to RNNs, *GPU* evaluation for BRNNs also produced marginal improvements, with 1.5 for *Kotlin*, 1.3 for *Java*, 1.2 for *Python*, 1.1 for *C++*, and *C#*, and performance on par for *JavaScript*.

The intrinsically parallelisable nature of CNN models resulted in significant speed-ups during *GPU* evaluation. For *Java*, *GPU* evaluation led to 26.8 times faster predictions, 21.6 for *C++*, 20.9 for *C#*, and 11.8 for *Kotlin*. However, average improvements were negligible for languages with larger average sizes, such as *Python* and *JavaScript*, with improvements of 1.2 and 1.1, respectively. Despite this, the per-token performances of *NN* models are equal for each language, and considering datasets with extremely large files, system integrators are anticipated to impose file size limits for code rendered in the browser.

Consistent with previous observations in the field [3], the *Pygments state-of-practice* resolvers demonstrate the shortest computation delays across all languages examined in this study, even when compared to the CNN-based models executed on *GPU*. However, this trend does not hold for the alternative *state-of-practice* resolver *TreeSitter*, which is found to be significantly slower than the *CNN GPU* inference, being on average 2.4, 5.0, and 2.6 times slower for *Java*, *C*++, and *C#*, respectively. Additionally, similarly to *Pygments*, the large average file sizes in languages like *Python* and *JavaScript* only slightly affect *TreeSitter*'s performance. *TreeSitter* manages to surpass the *CNN* on *GPU* models in the case of *Kotlin*, achieving an average speedup of 1.6 times, which translates to a marginal 0.6*ms* improvement.

In conclusion, while the architecture of *(B)RNN* models resulted in limited improvements in prediction delays, the proposed approach relying on *CNN*s empowers *GPUs* to exploit their parallelisable architecture consistently, achieving the best prediction delays attainable for on-the-fly syntax highlighting.

V. Related Work

The primary aim of the work presented here is to enhance the capabilities of real-time syntax highlighting tools by examining their generalisation abilities and offering improvements in evaluation speed. This research seeks to demonstrate the application of deep learning techniques to achieve not only effective but also efficient syntax highlighting. The subsequent section will outline the leading *state-of-the-art* methodologies that bear the closest relevance to the approach being proposed, and how these differ.

Type Inference. Deep learning has significantly influenced Type Inference, notably through *DeepTyper* [18], *Type4Py* [19], aiding the conversion from dynamically to statically typed languages. *DeepTyper* employs a sophisticated bidirectional *GRU* [6] framework, introducing a distinctive Consistency Layer to improve handling of long-range inputs, and utilises a softmax function to assign type probabilities to each token. Contrary to the approach in this research, which concentrates on analysing sequences of token rules, *DeepTyper* includes token identifiers to ascertain type names, a detail considered extraneous for the tasks at hand. *Type4Py* advances this concept with a more complex neural network architecture and necessitates costly preprocessing steps like AST derivation. However, the augmented complexity and processing requirements of models such as *DeepTyper* and *Type4Py* do not necessarily equate to enhanced performance for the methodology discussed here, highlighting a fundamental divergence in focus and efficacy between these models and the approach presented. TypeFix, leveraging advanced transformer technology [20], serves as a decoder network for lenient parsing and typing of *Java* code fragments, evolving from DeepTyper's foundation [21]. Its structure features a six-layer decoder, each layer enriched with multi-head attention and feed-forward mechanisms, enabling sophisticated handling of complex sequences. This design allows each layer's output to reflect a synthesis of all preceding unit combinations, facilitating the learning of generalizable input sequence patterns. The multi-headed attention further refines the model's capacity to discern intricate input relationships, surpassing traditional RNN models, which are limited by vanishing gradients [22]. Like the SH methodology and DeepTyper, TypeFix is trained using a synthetically created oracle, pairing *Java* token identifiers with their deterministic types, thereby predicting categorical probability distributions across a defined type vocabulary. Yet, this intricate architecture is not adopted for the immediate SH approach, mirroring the considerations for DeepTyper and Type4Py. The emphasis remains on crafting more streamlined and effective models for real-time SH, prioritising the unique demands and challenges of SH over the complexity offered by TypeFix.

Island Grammars. Island Grammars introduce a framework for grammar design, segregating grammar rules into "island" for specific subsequences and "water" for the remaining tokens [23]. This structure allows for targeted processing of sequences relevant for highlighting in SH tasks, with "island" rules focusing on highlight-worthy sequences and "water" rules managing the rest. Despite its potential, this methodology diverges from the current research's trajectory [3]. Crafting an island grammar demands a deep understanding of grammatical constructs and a meticulous definition process for productions, a task more intricate than creating a tree walker for existing grammars. This complexity contrasts with the current research's goal of simplifying development and improving automation in SH tasks. Moreover, island grammars do not fully resolve the challenge of processing incomplete language derivations, a limitation shared with the state-of-practice approach that this research intends to transcend. Hence, island grammars do not meet the aims of this research, which prioritises more streamlined and automated strategies for SH tasks.

Program Synthesis. The process described in this line of work also diverges fundamentally from the of Program Synthesis, which is concerned with the generation of programs that map inputs to outputs. Program Synthesis, exemplified by projects like *DeepCoder* [24] and *PQT* [25], aims to infer program structures to bind inputs with outputs, enhancing traditional search techniques through predictive neural networks. These models complement rather than replace search-based methods, focusing on program generation guided by input-output examples rather than understanding the intricacies of compilers or interpreters. Techniques such as execution-guided synthesis in *Execution-Guided Neural Program Synthesis* [26] aim to improve predictions based on program state manipulations, distinct from the operational semantics learning associated with compilers. Similar approaches enhance the synthesis or search process and rule generation, steering clear of mimicking compiler or interpreter functionalities [27], [28]. *NGST2* [29] introduces a formal method for program translation through trace-compatibility and cognate grammars, focusing on syntactic conversion between programming paradigms rather than delving into the mechanics of program execution akin to a compiler or interpreter. This highlights a clear distinction from the process presented in this work, which seeks to understand and replicate the mapping of inputs to outputs in the manner of compilers or interpreters, setting it apart from the objectives and methodologies of Program Synthesis.

VI. Conclusion and Future Work

This work advanced the domain of *On-the-Fly Syntax Highlighting*. It delivered an extended dataset to include six programming languages. Now including *C++*, *C#*, and *JavaScript*, in addition to the original *Java*, *Kotlin*, and *Python*, this expansion not only enriches benchmarking capabilities but also broadens the scope of application in diverse coding environments. The investigation into the generalisation capabilities of *state-of-the-art* RNN and BRNN methodologies has yielded promising results, demonstrating robustness with near-perfect accuracy and manageable time delays. The precision and efficiency demonstrated in benchmarking evaluations indicate that the CNN method not only maintains near-perfect predictions but does so at a significantly faster rate, especially when evaluated on *GPU* platforms. This positions the *CNN*-based approach as the front-runner in the realm of *On-the-Fly Syntax Highlighting*, both in terms of accuracy and speed.

Future work should also consider the efficiency of the training process. Preliminary investigations suggest that there is potential for reducing the number of training samples, which could lead to a significant decrease in training costs. Furthermore, the exploration of multilingual models would be a logical extension, potentially streamlining deployment in diverse programming environments and thereby increasing practical applicability. It is important to recognise that applying the principles of this research in other fields will inherently lead to improvements in both the development processes and the tooling. Employing this technology to recognise and interpret code snippets from diverse web sources, even those not strictly adhering to standard syntax, may boost how developers engage with code on various platforms. This will not only enhance tools for code comprehension and error detection but also refine the processes involved in developing these tools. Developers could shift their focus to creating straightforward, brute-force solutions, focusing less on performance optimisation or tolerance to noisy inputs. Simultaneously, the tools themselves are set to be more accurate and responsive. This dual advancement in both process and tooling promises a transformative impact on the software development lifecycle.

Assessing the impact of this novel tool stack for syntax highlighting, particularly how it has significantly transformed both development processes and the tools themselves, may also be a key focus for future work. The traditional reliance on complex systems of regular expressions has been entirely removed, with the potential of shifting the burden from developers to the tool, which now autonomously manages accuracy, generalisability, and performance. This shift suggests that it could be valuable to analyse the extent to which these changes may boost the production of syntax highlighters. Additionally, the development of reliable, accurate, and efficient syntax highlighters may now be more accessible, with a reduction in the expertise required. The a priori knowledge of the language grammar in all its details is no longer as critical, as AST walking provides a correct abstraction over these complexities. User studies may also consider evaluating the ability of users to produce accurate and efficient syntax highlighters for both novel and mainstream languages. This could further highlight the challenges faced by traditional regex-based systems, whose performance tends to deteriorate with increased grammatical coverage, especially as they would naturally attempt to approach the precision of brute-force resolvers. In terms of tooling, this work has also led to the development of new state-of-the-art syntax highlighters that are the most accurate and stable when dealing with incorrect derivations, a level of quality previously attainable only in local development environments. Therefore, future research should focus on user studies to explore the impact of this enhanced quality on the various online tasks that developers routinely engage in. These studies could extend beyond user preference or satisfaction to include the effects on tasks such as code reviewing, code searching, comprehension, and other activities involving code manipulation and understanding. By examining these factors, researchers could gain a deeper understanding of how these advancements influence the overall software development lifecycle, particularly in online and collaborative environments.

References

[1] A. Sarkar, "The impact of syntax colouring on program comprehension," in *Proc. Annu. Meeting Psychol. Program. Interest Group (PPIG)*, 2015, pp. 8–9.

[2] D. Asenov, O. Hilliges, and P. Müller, "The effect of richer visualizations on code comprehension," in *Proc. CHI Conf. Human Factors Comput. Syst. (CHI)*, New York, NY, USA: ACM, 2016, pp. 5040–5045, doi: 10.1145/2858036.2858372.

[3] M. E. Palma, P. Salza, and H. C. Gall, "On-the-fly syntax highlighting using neural networks," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2022, pp. 269–280, doi: 10.1145/3540250.3549109.

[4] M. E. Palma, A. Wolf, P. Salza, and H. C. Gall, "On-the-fly syntax highlighting generalisability and speed-ups - replication package," [Online]. Available: https://zenodo.org/records/14162905

[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2014, pp. 3104–3112.

[6] K. Cho et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1724–1734.

[7] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[8] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1243–1252.

[9] N. Ngoc Giang et al., "DNA sequence classification by convolutional neural network," *J. Biomed. Sci. Eng.*, vol. 9, pp. 280–286, Jan. 2016.

[10] T. Parrm, "ANTLR." 2022. Accessed: Jan. 29, 2024. [Online]. Available: https://www.antlr.org

[11] G. Brandl, "Pygments." 2022. Accessed: Jan. 29, 2024. [Online]. Available: https://pygments.org

[12] "Tree-sitter contributors," Tree-sitter. 2024. Accessed: Jan. 29, 2024. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[13] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Accessed: Sep. 08, 2024. [Online]. Available: https://www.tensorflow.org

[14] D. C. Montgomery, *Design and Analysis of Experiments*. Hoboken, NJ, USA: Wiley, 2017.

[15] A. Vargha and H. D. Delaney, "A Critique and Improvement of the "CL" Common language effect size statistics McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

[16] S. C. Seow, *Designing and Engineering Time: The Psychology of Time Perception in Software*. Reading, MA, USA: Addison-Wesley Professional, 2008.

[17] J. Dabrowski and E. V. Munson, "40 years of searching for the best computer system response time," *Interact. Comput.*, vol. 23, no. 5, pp. 555–564, 2011.

[18] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2018, pp. 152–162.

[19] A. M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4py: Practical deep similarity learning-based type inference for python," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 2241–2252, doi: 10.1145/3510003.3510124.

[20] A. Vaswani et al., "Attention is all you need," in *Proc. Conf. Neural Inf. Process. Syst. (NIPS)*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.

[21] T. Ahmed, P. Devanbu, and V. J. Hellendoorn, "Learning lenient parsing & typing via indirect supervision," *Empirical Softw. Eng.*, vol. 26, no. 2, pp. 1–31, 2021.

[22] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[23] L. Moonen, "Generating robust parsers using island grammars," in *Proc. Work. Conf. Reverse Eng. (WCRE)*, 2001, pp. 13–22.

[24] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," 2016, *arXiv:1611.01989*.

[25] D. A. Abolafia, M. Norouzi, J. Shen, R. Zhao, and Q. V. Le, "Neural program synthesis with priority queue training," 2018, *arXiv:1801.03526*.

[26] X. Chen, C. Liu, and D. Song, "Execution-guided neural program synthesis," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 3–6.

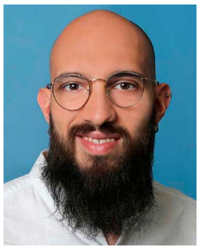
[27] S. Alford et al., "Neural-guided, bidirectional program search for abstraction and reasoning," in *Proc. Complex Netw. & Their Appl.,* vol. 1, Berlin, Hiedelberg: Springer International Publishing, 2022, pp. 657–668.

[28] E. Noriega-Atala, R. Vacareanu, G. Hahn-Powell, and M. A. Valenzuela-Escárcega, "Neural-guided program synthesis of information extraction rules using self-supervision," in *Proc. 1st Workshop Pattern-Based Approaches to NLP Age Deep Learn.*, 2022, pp. 85–93.

[29] B. Mariano, Y. Chen, Y. Feng, G. Durrett, and I. Dillig, "Automated transpilation of imperative to functional code using neural-guided program synthesis," in *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022, pp. 1–27.

Marco Edoardo Palma received the M.Eng. First-Class Honours degree in computer science with artificial intelligence from the University of Southampton, U.K., in 2021. He is currently working toward the Ph.D. degree with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland. His research explores the development of artificial intelligence tools and strategies to enhance software engineering processes and tools. Currently, his work centres on the deep abstraction strategy, which automatically compiles algorithms with high space and time complexity into efficient statistical models.

Alex Wolf received the master's degree in software systems from the University of Zürich, in 2022. He is currently working toward the Ph.D. degree with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland, where their research focuses on advancing machine learning, software architecture, and engineering. Prior to beginning their doctoral studies, he gained industry experience through various software engineering roles, providing a strong foundation in practical and technical problem-solving. His research interests lie at the intersection of machine learning and software engineering, with a particular focus on practical applications and the integration of blockchain technology.



Pasquale Salza received the Ph.D. degree in computer science from the University of Salerno, Italy. He is a Senior Research Associate with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland. His research interests include software engineering, machine learning, cloud computing, and evolutionary computation.

Harald C. Gall (Member, IEEE) is a Professor of software engineering and the Director of the Software Evolution and Architecture Lab (s.e.a.l.), Department of Informatics, University of Zurich, Switzerland. He held visiting positions with Microsoft Research, USA, and University of Washington in Seattle, USA. His research interests include software evolution, software architecture, software quality, and green software engineering. He has worked on developing new ways in which data mining of software repositories and machine learning can contribute to a better understanding and improvement of software development.