Exploration of Convolutional Neural Network models for source code classification

Exploration of Convolutional Neural Network Models for Source Code Classification

Francesco Barchi[a,*], Emanuele Parisi[a], Gianvito Urgese[b], Elisa Ficarra[b], Andrea Acquaviva[a]

[a] *Università di Bologna (DEI), Viale del Risorgimento, 2 - 40126 Bologna, Italy*
[b] *Politecnico di Torino (DAUIN), Corso Duca degli Abruzzi, 24 - 10129 Torino, Italy*

Abstract

The application of Artificial Intelligence is becoming common in many engineering fields. Among them, one of the newest and rapidly evolving is software generation, where AI can be used to automatically optimise the implementation of an algorithm for a given computing platform. In particular, Deep Learning technologies can be used to the decide how to allocate pieces of code to hardware platforms with multiple cores and accelerators, that are common in high performance and edge computing applications. In this work, we explore the use of Convolutional Neural Networks (CNN)s to analyse the application source code and decide the best compute unit to minimise the execution time. We demonstrate that CNN models can be successfully applied to source code classification, providing higher accuracy with consistently reduced learning time with respect to state-of-the-art methods. Moreover, we show the robustness of the method with respect to source code pre-processing, compiler options and hyper-parameters selection.

Keywords: Deep Learning, LLVM-IR, Code Mapping, Heterogeneous Platforms

1. Introduction

While Artificial Intelligence (AI) is now widespread in many engineering fields, a novel and rapidly evolving application is software generation and in particular compiler optimisation [1]. Considering the increasing complexity of High Performance Computing and Edge Computing platforms adopted in research and industrial applications, the problem of intelligently and automatically optimizing software on these platforms is one of the major challenges for software developers, preventing their fast and easy exploitation on the field [2].

*Corresponding author
Email address: francesco.barchi@unibo.it (Francesco Barchi)

One can view this AI application as a technology helping itself. This help is needed to decide how to efficiently map a given algorithm to a target platform featuring multiple cores and accelerators such as GPUs, DSPs and custom hardware accelerators on FPGA. AI can support many aspects of this problem, such as parallelization, compiler flag optimisation, code transformation [1]. In this paper, we focus on the kernel allocation decision, that is where it is better to allocate a piece of code or a function (from now on called a computational kernel), either on the main processor (CPU) or one of the available accelerator (e.g. GPU, DSP), to optimise a given metric (e.g. overall execution time).

Without AI support, this decision requires porting of the computational kernel to the target accelerator and its profiling, thus imposing coding and deployment effort for the developer.

To face this issue, code analysis techniques based on deep learning methods have been developed for high-level languages such as OpenCL [3]. To improve the generality of this approach, these methods have been applied to machine-independent languages, like the *Low Level Virtual Machine* (LLVM) *Intermediate Representation* (IR) [4] by recent academic papers [5–8]. The advantage of LLVM representation is its hardware-independence and generality so that it can be reached from different high-level languages.

A deep learning code classifier takes as an input a kernel in a given code representation (e.g. IR), and after a preliminary code transformation, it implements a language model based on a deep learning network. This language model extracts the relevant code features needed by the final dense network layer to take the kernel allocation decision (e.g. the input kernel runs faster in GPU or CPU).

In this work, we present a method for classification of LLVM source code (we refer to it as *DeepLLVM*) where we introduce the use of Convolutional Neural Networks (CNN) in code classification. Indeed, while CNN have been used for source code analysis [9] their use for source code classification has never been explored, as approaches using recurrent networks such as LSTM are more common in source code analysis for code classification [3, 5–7, 10]. We believe that, given the popularity of CNN and their interesting properties such as fast learning time and availability of design and configuration tools, an evaluation of their usage for source code analysis is of relevance.

Our CNN implementation is a one-dimensional convolutional layer (Conv1D) integrated into *DeepLLVM* and using a final global max-pooling (GMP) layer to extract knowledge from syntactic language elements (tokens) of a kernel compiled in IR.

DeepLLVM is divided into two modules: i) The source code preprocessing module that identifies the most significant syntactic elements and reducing them to a sequential list of integers. ii) The language classifier component trained using a supervised learning method that includes the Conv1D CNN model and where we also integrated an LSTM model used in state-of-art papers for comparison.

We trained the network using a dataset of OpenCL kernels whose execution time has been profiled on CPU and GPU [3]; then we evaluated the classifi-

cation accuracy in mapping each kernel to the best compute unit. We report the results of a comparative analysis of CNN-based method comparing with LSTM to demonstrate the effectiveness of the CNN model. We assess its robustness with respect to network and training hyper-parameters through extensive exploration, and we evaluate the impact of code preprocessing steps on the classification accuracy.

Besides the comparison of the network model, we also compared our method against state-of-art approaches working on IR but exploiting different methods for code information extraction. While we believe that approaches exploiting code flow graph information are promising [7, 8, 10], we show that our method provides overall better results in terms of training time, classification accuracy and speedup.

We finally discuss the impact of some characteristics of the dataset used by almost all recent works on this topic [3] and their impact on the validation methods adopted.

The rest of the paper is organised as follows: In Section 2, we describe the research area and the relevant related works. In Section 3, we describe our methodology: the source code preprocessing module, the code classifier, and the hyper-parameters exploration strategy. In Section 4, we assessed the performance of several CNN model configurations and provided comparisons against RNN model with the variation of kernel compilation options and token filtering strategies. Then, in Section 5, we provide some conclusions and future works.

2. Background and Related Works

In literature can be found a rapidly growing research tries to answer the question *can we analyse the code like text?* exploiting natural language translation, classification and code modelling techniques for code quality assessment [9], plagiarised source code detection [11] or classification for execution on a certain hardware target [3, 5–7, 10]. As shown in the survey of Allamanis [12], the source code maintains some properties of natural languages (it can be considered, like text, a human communication form) but it has profound differences. Some of the code properties like executability, formality and structure make it more complex to analyse than text. Compiler designers started considering the adoption of machine learning techniques to obtain heuristic compilers capable of learning from the data [1, 13] for code optimisation.

Research in the field of Natural Language Processing (NLP) has evolved considerably in recent years. The current SOTA is composed by the language models BERT [14], XLNet [15] and GPT [16]. While BERT uses Autoencoders and token-masking to generalise model knowledge, XLNet and GPT use self-regressive models based on transformer networks [17]. Different versions of GPT-3 composed of a variable number of weights (from 3B to 175B) are able to solve problems of text generation, question answering, reading comprehension in zero-shot, one-shot and few-shot mode. Recently the ability of GPT-3 to generate JavaScript XML (JSX) code in few-shot (two sample context) mode

has been shown. These techniques are the cutting edge of research on generic text understanding models, but their size does not make them easily applicable in domain-specific contexts.

Several domain-specific techniques, instead, have been proposed in the literature to represent programs using a set of quantifiable properties or features compatible with the inputs of the learning module [18]. Standard machine learning algorithms typically work on fixed-length inputs, so the selected properties shall be transformed into a fixed-length vector of features (boolean, integer, or real values). Compiler researchers have designed, during the years, various forms of program features for machine learning algorithms. These include static code structures extracted from the source code or the compiler intermediate representation [19] and dynamic profiling information obtained through runtime profiling of the program execution [20]. Compiler optimisation methods based on supervised learning have been proposed using Bayesian Networks [21], Support Vector Machines [22, 23], Decision Trees [18, 24] and Graph Kernels [25].

In the last decade, the problem of deciding the most suitable hardware unit on which to execute a given computational kernel raised with the increasing complexity and heterogeneity of digital platforms. Source code analysis can help this decision avoiding to make useless porting efforts. Moreover, a profiling approach based on source code analysis without the need for the final target hardware or an accurate virtual (simulation or emulation) platform can speed up the embedded systems development process.

Along this direction, in 2013, Grewe [26] developed a workflow to translate an OpenMP program in OpenCL and to decide for each generated OpenCL kernel the most suitable compute unit between CPU and GPU. There, the authors defined metrics to extract from the code (like the number of calculation operations or local and global memory access) to make decisions based on a probabilistic method (i.e. a decision tree classifier).

In last years, a research line exploiting the maturity of deep learning methods has started since the work of Cummins et al. [3], where the decision tree classifier was replaced with a deep learning model based on a RNN. Thanks to deep learning, it is no longer needed to extract the features manually since they are inferred automatically during the training phase and improves classification accuracy compared to [26].

The methodologies proposed in [26] and [3] were developed and customised for kernels implemented in OpenCL, thus constraining the methodology to work with a given source programming language. To overcome this limitation, Ben-Nun et al. [6] and Barchi et al. [5] introduced the adoption of code analysis at the intermediate representation (IR) level of the LLVM compiler. LLVM is increasingly adopted in the embedded system world, because it is capable of decoupling the front-end compiler from the target architecture, in this way many optimisation steps can be performed at the IR level before generating the binary machine code. Source code features, at this intermediate level, can be exploited to perform complex compilation decisions, including allocating code fragments to architecture devices. Machine learning techniques can be applied

to learn these characteristics by creating a learning model based on training code fragments.

The LLVM based methods presented in [5] and [6] differ for the strategy for the projection of source code in the continuous metric space. In [5] the code stream is filtered and then introduced directly into the network, relying on the Embedding Layer for the learning of the best token projection. On the other side, in [6], the authors propose Inst2Vec a system to pre-train the embedding layer analysing the Contextual Flow Graph (XFG).

Further contributions to this projection problem have been devised later. In Kheerthy et al. [7], a procedure to project an IR in a continuous metric space directly, called IR2Vec is proposed. In Cummins et al. [10], the authors propose *ProGraML*, an extension of [6] where a GNN-based classifier is proposed for the first time. Independently, in Brauckmann et al. [8] another GNN-based classifier was proposed able to learn vertex embeddings by itself. Moreover, in [8] the GNN is used to analyse both an LLVM-IR Control and Data Flow Graph (GNN-CDFG) and a Clang Abstract Syntax Tree (GNN-AST).

Concerning the deep neural network model, all previous work use RNNs, that have been introduced to process temporal sequences [27, 28]. An RNN maintains an internal state, acting as a memory, that summarises the information extracted from the input sequence.

Very successful implementation of RNN is the Long Short-Term Memory (LSTM), a network able to learn when to memorise or forgot information of the input sequence and correlate together elements at different times. For this reason, LSTM is the model adopted in state-of-art papers [3, 6, 7].

However, given the widespread use of CNN in the context of image recognition [29] but also in NLP [30] as well as fast learning time and the maturity of network design and configuration tools, it is worth exploring their application to source code classification.

Behind the success of this type of network, there is the assumption of information locality in the input data. All data inside a region called "kernel" are considered correlated, and this correlation is weighed by a filter, identical for any region considered in the input. For image classification, the kernel shape has two dimensions, but this technique can also be used in temporal signals using one-dimensional kernels. This is the approach we follow in this work, where we give as an input to the CNN a tokenised and filtered code stream directly without using additional information.

The convolution operation performs an element-wise multiplication between the input data in the kernel region and the filter and accumulates the results in a single scalar. The filter moves along all input dimensions by a fixed step called stride. A convolution layer uses multiple filters to explore a different type of kernel relationship. Each filter contributes to building a channel of the output tensor. In this work CNN is for the first time introduced in a code classifier method, fully characterising its performance with an extensive exploration and comparison with LSTM. To make this comparison fair, we implemented the LSTM model in our *DeepLLVM* framework.

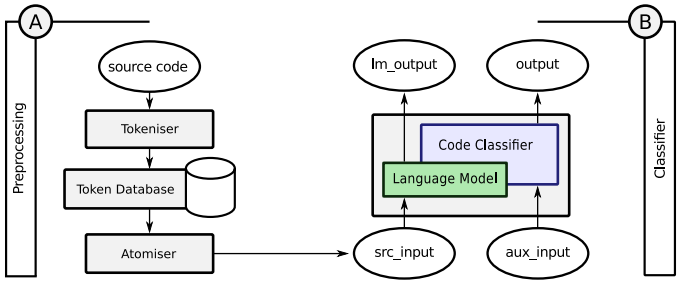

Figure 1: *DeepLLVM* flow representation of operations to be performed for the construction of the code classifier. A) Source code preprocessing steps for inserting the code into the classifier B) Black-box representation of the classifier

3. Methodology

In this Section, we present the CNN-based code classification methodology, that we included in the *DeepLLVM* tool working on IR. We also integrated an LSTM classifier and we performed an hyper-parameters exploration on both DNN models to make a solid comparison and evaluate the impact of code preprocessing. We will then evaluate techniques and models taken from the Natural Language Processing (NLP) research area. The aim is to use token-based methods that are easy to use like convolutional networks (CNN) and to evaluate their performance on LLVM-IR, a language that is very unrelated to natural language. In addition, the stability of the model has been explored with different methods of initialisation and hyperparameters.

In the rest of this Section we present the *DeepLLVM* flow in which the CNN has been integrated. Referring to Figure 1, *DeepLLVM* is composed of two steps: i) Source code preprocessing (Section 3.1), which identifies the most significant syntactic elements (Tokenisation) and translates them to a sequential list of integers that are further filtered based on their relevance in the source code (Atomisation). ii) Code classification (Section 3.2), which exploits a Neural Network based on layer models[1] trained using a supervised learning method. We then describe the methodology for exploration of hyperparameters (Section 3.3), and the metrics adopted for evaluation (Section 3.4).

3.1. Code Preprocessing

Since machine learning models work with numerical data, a procedure to convert source code into a form suitable to be processed by the input layer of the considered models is typically applied. We start from a dataset of source code written using a high-level programming language. Then we compile dataset elements (i.e. computational kernels) using a compiler able to emit LLVM-IR. To this purpose we used the *clang* compiler that allows compiling C-like languages

[1]Convolution Layer, Recursive Cells, Dense Layer and Max Pooling

LLVM-IR Code Fragment

```
%9 = and i64 %8, 4294967295
%10 = getelementptr inbounds <4 x float>, <4 x float>* %1, i64 %9
%12 = fsub <4 x float> <float 1.0e+00, float 1.0e+00, float 1.0e+00, float 1.0e+00>, %11
%13 = fmul <4 x float> %11, <float 3.0e+01, float 3.0e+01, float 3.0e+01, float 3.0e+01>
```

Tokenisation

```
_local = and i64 _local , _integer_constant
_local = getelementptr inbounds _float_4 , _float_4 * _local , i64 _local
_local = fsub _float_4 _vector_constant , _local
_local = fmul _float_4 _local , _vector_constant
```

Atomisation

```
10 11 13 14 10 12 15
10 11 16 17 18 12 18 19 10 12 14 10
10 11 20 18 21 12 10
10 11 22 18 10 12 21
```

Figure 2: **Example of code transformations:** The code in the top pane is an LLVM-IR code fragment. The code in the middle pane contains the result of the transformations applied in the tokenisation phases. The tokens sequence in the bottom pane is the network input, the result obtained after the atomization phase.

(C, C++, Objective C) and OpenCL code for *nvptx* (NVidia), *amdgcn* (AMD) and *spir* (Standard Portable Intermediate Representation) architectures.

The LLVM-IR code obtained after the compilation undergoes a preprocessing step before being fed into the neural network (Fig.1-(A)). In this work, we adopt a slightly modified technique than the one we adopted in in [5].

The ***Tokenisation*** procedure identifies the most significant language syntactic elements (tokens) within the sequences. Tokens are catalogued and placed in a dictionary. Then, the ***Atomisation*** procedure transforms code sequences replacing the characters that compose a token with the integer identifier of the token in the dictionary and performs a filtering procedure to select the most informative tokens.

The *Tokenisation* procedure is organised in two steps. The *pre-tokenisation* phase acts on each line of a kernel and performs the following operations:

- Remove empty lines and comments.
- Remove all lines outside the function body.
- Replace vector and array data-types with a simplified version.
- Replace vectors, arrays and float constants with a placeholder maintaining the type and removing the immediate value.
- Insert a space before and after the symbols
  ( ) [ ] { } < > = * : ,

During this phase, the procedure simplifies complex data types and replaces constants with placeholders, obtaining a significant reduction of the code fragment length. For example, LLVM can express real constants in different ways: i) standard decimal notation (e.g. 6.563989), ii) exponential notation (e.g.

1.179029e+45), iii) hexadecimal notation (e.g. 0x5612585f32165865). These representations are replaced with a placeholder ("_float_constant"). After *pre-tokenisation*, it is possible to identify as a token every sequence of characters separated by spaces.

The *post-tokenisation* transformations act directly on the tokens for applying the following higher level generalisations:

- Remove unnamed meta-data (tokens starting with (!)) and attribute groups (tokens starting with (#)).
- Replace variable and function names with a placeholder.
- Identification of special labels starting with "phi", "pre", "in", "preheader" and "loopexit".
- Identification and transformation of integer constants.
- Identification and transformation of global and local unnamed identifiers (e.g. %5 → _local, @16 → _global)

The tokens are then replaced with a unique integer identifier using the same approach proposed in [31]. Figure 2 provides an example of the preprocessing pipeline we use.

Because the performance of text-based deep-learning systems heavily depends on the dictionary chosen to transform the input into a numerical sequence [32], a final filtering procedure is applied before the token sequence is given as input to the neural network. Long sequences require many training samples and complex models capable of storing and correlating information for more extended periods. A common practice in the Natural Language Processing (NLP) field consists in removing less-informative tokens [33] for decreasing the mean length of the sequences to be analysed and reducing the burden of correlating distant tokens.

Here a weight function is used, namely the *term frequency - inverse document frequency* (*Tf-Idf*), to identify the less-informative tokens. The *Tf-Idf* can be obtained as in Eq.1

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) * \overbrace{ln\left(\frac{1}{\text{df}(t, D)}\right)}^{\text{idf}} \tag{1}$$

Given a dataset $D : (d_1, d_2, \ldots, d_n)$ (corpus of documents), a document $d : (t_1, t_2, \ldots, t_n)$ (sequence of tokens) and a token t, the *Tf-Idf* is the product between the term-frequency (tf) and inverse-document-frequency (idf). The idf is the natural logarithm of the inverse of document-frequency (df).

$$\begin{aligned} \text{tf}(t, d) &= \frac{|\{t' : t' = t, \forall t' \in d\}|}{|d|} \\ \text{df}(t, D) &= \frac{|\{d' : t \in d', \forall d' \in D\}|}{|D|} \end{aligned} \tag{2}$$

The term-frequency is the ratio between the occurrences of term t in a sequence of tokens d and the length, in terms of tokens count, of d. The document-frequency is the ratio between the number of sequences of tokens where the

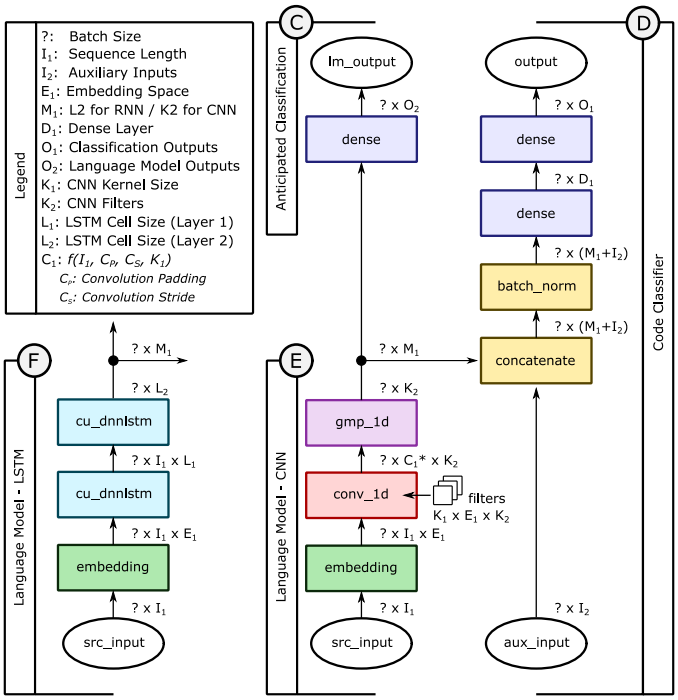

Figure 3: Informal representation of the classifier structure. The figure represents the layers of the classifier as rectangles identified by a label describing the operation performed. The arrows indicate the movement of data (tensors) whose dimensions are made explicit by symbolic values described in the legend. E) CNN-based language model F) RNN-based language model C) Early classification for specific language model training D) Overall classification network, takes into account the source code and context data flow.

token appears and the total number of sequences. The idf reduces the term-frequency value for very common tokens, and increase the term-frequency for specific tokens contained in few sequences. Applied to source code analysis, each document d represents a kernel, while the corpus of documents D represents the entire dataset of benchmarks.

We can use this weight to build a token-blacklist to delete the tokens with a low informative contribute from all kernels. The token-blacklist can be used globally, or we can use the *Tf-Idf* score to create fine-grain filters removing from each sequence only the tokens with a poor local score.

3.2. CNN and LSTM based classifiers

The code classifier takes the output from the *Atomiser* as shown in Figure 1-(B). The input is a tuple containing the preprocessed source code fragment (src_input), and auxiliary data that define the context of usage of the source code (aux_input). We use the same auxiliary inputs included in the dataset [3].

Internally, the classifier is a neural network divided into two components: language model and features classifier. The language-model is in charge of

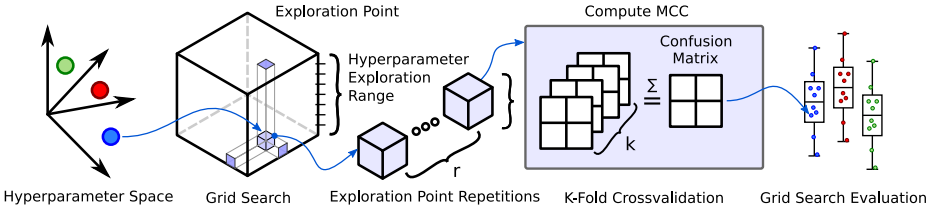

Figure 4: Schematic representation of Hyper-parameters exploration. Within the hyper-parameters space some points are chosen to be explored by means of a grid-search. For each hyper-parameter an exploration range is identified (Hyper-parameter Exploration Range). Each point is explored r times. Each exploration includes a k-fold cross-validation and training of k classifiers. The confusion matrices of the classifiers in cross-validation are added together, then the ACC and MCC are calculated. To evaluate the hyper-parameters space, each exploration point is seen as the distribution of MCC and ACC values of its repetitions.

reducing the token sequence into a point in $\mathbb{R}^{M_1}$. The features classifier analyses the output of the language model.

The two outputs are the features-classifier output and the language-model output. Both outputs can be independently used, and each one is associated with a loss-weight. The loss weight is a scalar coefficient that defines the output loss contribute over the global-loss score.

The network structure proposed in this work is depicted in Figure 3. The figure shows the main flow based on the CNN language model network (Figure 3-(E)) and the LSTM that we used for comparison (Figure 3-(F)).

The network input ("src_input") is a tensor composed of batch-size sequences, each one composed of sequence-length (I_1) elements. We will refer to the input elements with the term *token-indexes* since each component represents the position of a token inside the token dictionary. Since the following layers need to work on comparable data, and the token indexes do not have this property because we cannot define a distance metric between two indices, the sequence of token-indexes must be projected into a metric space.

The Embedding layer is the first layer of the network that receives sequences of token-indexes and projects each element into an embedding space $\mathbb{R}^{E_1}$. The output of the Embedding Layer is, therefore, a list (of length batch-size) of sequences (of length I_1) where each token-index has been transformed into a E_1-dimensional vector in the embedding space. The weights of the Embedding Layer determine how the token-indexes are projected in the embedding space. At the beginning of the training, the projection in the embedding space starts in a random condition.

The purpose of the language model is to process the output of the Embedding Layer (a vector in $\mathbb{R}^{I_1,E_1}$) and, at the end of the process, reduce each sequence to a single point in the features space ($\mathbb{R}^{M_1}$). Features can now be passed to the second part of the network (Fig. 3-(D)) that will perform the classification.

Our CNN implementation of the language model consists of a one-dimensional convolution layer, followed by a global max-pooling layer. The convolutional layer applies in the sequence direction a number K_2 of filters of size $K_1 x E_1$ so

as to process each sequence produced by the Embedding Layer. The global max pooling acts for each channel of convolution layer output and selects the maximum value. This structure is inspired by the one adopted in [34] for sentence sentiment classification.

The LSTM model is composed by two layers as in [5] and [3]. The first layer elaborates the input sequence and produces another sequence in output. The second layer elaborates the output of the first layer maintaining only the last output element, considering it a point in the feature space.

3.3. Hyper-parameters exploration

The goal of the Hyper-parameters exploration is twofold: i) To assess the robustness of the method w.r.t. network and training hyperparameters; ii) To evaluate the impact of source code preprocessing parameters.

The exploration is done with a multi-stage grid-search. We divided the hyper-parameters into three categories:

- *Network* hyper-parameters
- *Training* hyper-parameters
- *Dataset* hyper-parameters

The *network* hyper-parameters are specific to the network model that we consider. They are divided into CNN and LSTM hyper-parameters. These two sets have in common the parameters that define the sequence input length (I_1) the output size of the embedding-layer (E_1) and the output size of the last dense-layer (D_1).

The CNN hyper-parameters are the kernel size (K_1) and the number of filters (K_2). The LSTM hyper-parameters are the cell-size of the first LSTM layer (L_1) and the cell-size of the second LSTM (L_2) layer.

The *training* hyper-parameters are instead specific to the supervised training method: i) the chosen training algorithm (SGD, Adam) [28]; ii) training algorithm hyper-parameters (specific for the training algorithm); iii) batch size (1, 16, 32, 64) and loss weight (0.0, 0.1, 0.2, ..., 1.0) of the learning model output (*lm_output*).

The exploration of *dataset hyper-parameters* is used to evaluate the impact of source code preprocessing options, namely: i) Padding strategy (add a null token at the end or before the sequence); ii) Truncating strategy (keep the initial or final tokens); iii) Token filtering policy in order to eliminate tokens with a low information contribute (token blacklist, *Tf-Idf* threshold); iv) Compilation optimisation options (`-O0`, `-O1`, `-O2`).

The evaluation process is depicted in Figure 4. The exploration is composed of a first grid search on *network hyper-parameters* followed by a second grid search on *training hyper-parameters*. The grid-searches explore a subset of the hyper-parameters space. For each hyper-parameter, we decide a set of values of interest that define its exploration range.

The exploration point is a configuration of hyper-parameters, and it is evaluated multiple time in order to take into account the intrinsic model variability during the training process and improve the statistic confidence of the process. The training of an exploration point repetition is performed in cross-validation. The whole dataset is split in k subsets, and in turn, we train k classifiers (fold-classifiers) changing the subset used as test-set. The confusion matrices of each fold-classifier are reduced using a sum operator.

3.4. Metrics

As for accuracy metrics, we consider both the Accuracy (ACC) and the Matthews Correlation Coefficient (MCC[2]). MCC has been introduced because it has shown to be effective, especially in the presence of unbalanced datasets [35], like the one we consider in our experiments.

$$C = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{MCC} = \frac{ad - cb}{\sqrt{(a+b)(a+c)(d+b)(d+c)}} \tag{3}$$

Given a confusion matrix $C \in \mathbb{R}^{2,2}$ the MCC is a metric that contrary to the Accuracy (ACC) considers the whole classifier behaviour and provides more consistent results when dataset labels are not balanced. Equation 3 reports the formula for computing MCC. Variables a and d represents true positives and true negative while b represents false negatives and c represents false positives.

All exploration points can be compared using the mean and the standard deviation of their performance distributions. An exploration point is evaluated given the performance (MCC and ACC) distribution of its repetitions.

Accuracy (ACC) and Matthews correlation coefficient (MCC) are suitable for a general assessment of classifier performance. However, in our case, the classification purpose is to minimise the execution time on a given platform by selecting the best compute unit. A similar consideration can be done for other metrics (e.g. energy, power).

We measure the impact of wrong decisions made by the classifier using the speedup with respect to fixed mappings (i.e. all kernels in the same compute unit). These metrics can be obtained during the labelling process of the classifier. Concerning the present work, the dataset collected in [3] was labelled with the compute unit showing the best runtime performance evaluated with the execution of OpenCL kernels. The tested devices for this dataset were a CPU, AMD-GPU, and Nvidia-GPU processors.

In Algorithm 1 we report a simplified summary of all operations described so far using a pseudo-code formalism. The reported steps describe the transformation of the dataset expressed in high-level source code in a token flow until obtaining the results of the exploration of the hyperparameters of the CNN model described in this section.

[2]MCC varies between -1 and +1, being +1 the best result possible.

```
input  : High level source code dataset (eg. OpenCL)
output: Best parameters and performances of LLVM-IR classifier
dataset ← new list;
foreach src in dataset_opencl do
    llvmir ← compile_opencl(src);
    llvmir_tokens ← new list;
    foreach line in llvmir do        // section 3.1, tokenization
        line ← simplify_line(line);
        foreach token in line do
            token ← simplify_token(token);
            if token is valid then llvmir_tokens ← add token;
        end
    end
    dataset ← add llvmir_tokens;
end
dataset ← atomize(dataset)           // section 3.1, atomization
if tfidf_threshold then                  // section 3.1, filtering
    dataset ← filter_tfidf(dataset)
end
network_gs_results ← new list ;
foreach cnn_conf in network_gs do          // sections 3.3, 3.4
    cnn_conf_results ← new list ;
    foreach i ← 1 to n_repeat do
        cnn_conf_results ← add cnn_train_xvalidation(cnn_conf);
    end
    network_gs_results ← add evaluate_metrics(cnn_conf_results);
end
network_gs_best ← get_best_conf(network_gs_results);
training_gs_results ← new list ;
foreach cnn_conf in training_gs do          // sections 3.3, 3.4
    cnn_conf_results ← new list ;
    foreach i ← 1 to n_repeat do
        cnn_conf_results ← add cnn_train_xvalidation(cnn_conf);
    end
    training_gs_results ← add evaluate_metrics(cnn_conf_results);
end
training_gs_best ← get_best_conf(training_gs_results);
```

Algorithm 1: Whole system pseudocode

Dataset	Device	CPU	GPU
AMD	Tahiti 7970	400 (58.8 %)	280 (41.2 %)
NVIDIA	GTX 970	293 (43.1 %)	387 (56.9 %)

Table 1: Labels distribution in the source dataset. For both GPU considered a slight 60%/40% unbalance in labels assignment can be observed.

4. Results & Discussion

In this Section, we report the results of the exploration of CNN source code classification method to assess the robustness w.r.t. hyper-parameters and we assess the impact of code preprocessing. We then compare CNN and LSTM models in terms of classification accuracy, training time and speedup of kernel execution compared to a fixed mapping. Finally, we compare for completeness our approach with recent state-of-art deep learning solutions working on LLVM but exploiting different language information, such as CDFG. This comparison does not provide insights on the effectiveness of the CNN itself. However, we show that our results are comparable or better than state-of-art methods in terms of accuracy, speedup and training time.

Section 4.1 describes the main properties of the kernel dataset, as well as the preprocessing and filtering operations we applied for producing the symbol sequences fed into the machine learning models. Section 4.3 details the hyper-parameters optimisation procedure we used for devising an appropriate CNN architecture. Section 4.4 provides exhaustive comparisons between CNN and RNN and explores how token filtering impacts on classification accuracy. Then, it provides details regarding the time required for training the two architectures with different input and batch sizes. Section 4.5 describes classifier performance taking into account runtime and speedup. Section 4.6 discusses the comparison with other LLVM approaches. Finally, Section 4.7 summarises the findings coming from the experiments analysis we performed.

4.1. Dataset description

For training and testing with *DeepLLVM*, we used a composition of OpenCL kernels coming from six source code collections [3]. Each element of the dataset has a label denoting the best performing computation device between a CPU and a GPU (AMD Tahiti or Nvidia GTX). The authors of the dataset executed each kernel using different load (byte transfer) and different level of parallelism (workgroup size), keeping track of the time required for executing each kernel on the available devices. Each triple is composed of: a kernel, byte transfer size and workgroup size and it is labelled with the device exposing the best runtime performance. The full dataset is composed of 680 triples and 256 different kernels, and it is characterised by a slight unbalance in labels assignment, detailed in Table 1.

OpenCL kernels stored in the source dataset are not suitable to be classified as they are, because machine learning models detailed in Section 3.2 require

Grid search	Hyper-parameter	Values
Network	Input size	1024, 2048, 4096
	Padding/Truncating strategies	pre, post
	Embedding size	64, 128
	Conv kernel size	5, 7, 9
	Conv kernel number	32, 64, 128
	Dense layer size	64, 128, 256
Training	Batch size	16, 32, 48, 64
	Aux output weight loss	0.0, 0.1, ..., 1.0

Table 2: Range of hyper-parameters values tested during the network and training grid-searches.

a sequence of numerical symbols as input. First of all, input sources were translated into LLVM intermediate representation running clang (v7.0.1) on each input kernel. We created a second experiment trunk adding the `-O2` flag to the clang compiler command line for checking whether any middle-end code transformation impact on classification accuracy. Moreover, we produced a third kind of sequences produced tokenising OpenCL kernels as they are for the sake of comparing our preprocessing pipeline with the one of [3].

4.2. Token filtering settings

We used two methods for cleaning the input sequences from symbols with poor informative content:

- Blacklist filtering: removing a set of intermediate representation tokens from each kernel.
- *Tf-Idf* filtering: each token-kernel pair *(t, d)* is assigned a score as detailed in Section 3.1. Whenever the score of *(t, d)* is lower than the given threshold, occurrences of token t are removed from document d.

Concerning blacklist filtering, we produced a list of common tokens that we assumed not to be strongly informative for kernel-device mapping.

In *Tf-Idf* filtering, it is crucial to devise a score threshold for removing only redundant tokens. Since such thresholds are dataset dependent, and no previous literature works provide methods for computing them, we evaluated the distribution of the average *Tf-Idf* score per document and sampled four values from it. Figures 5 and 6 show how the average *Tf-Idf* score per document distributes in the two LLVM datasets. Both distributions share the same shape and highlight a peak around 0.005 with a small tail up to 0.035 and 0.025. We chose to test four *Tf-Idf* scores around the most common average values, specifically 0.001, 0.003, 0.006 and 0.008.

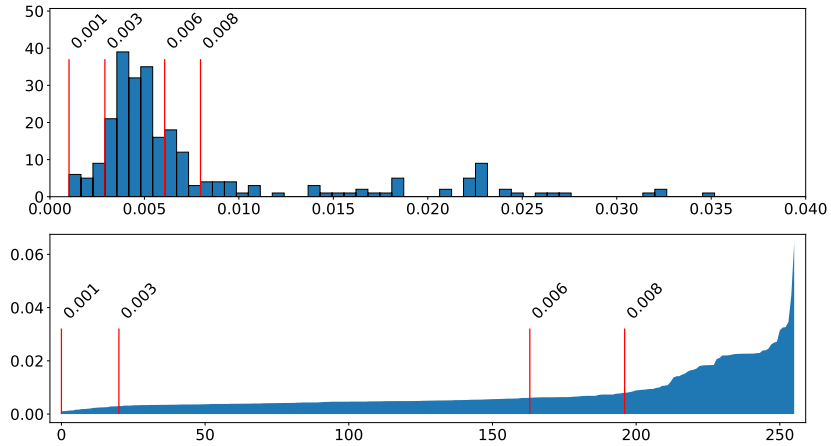

Figure 5: *Tf-Idf* analysis applied on LLVM `-O0` dataset. Distribution of the average *Tf-Idf* score measured per document (top). Average *Tf-Idf* score directly represented in a bar-plot. In the figure are depicted the score threshold used in *Tf-Idf* filtering evaluation.

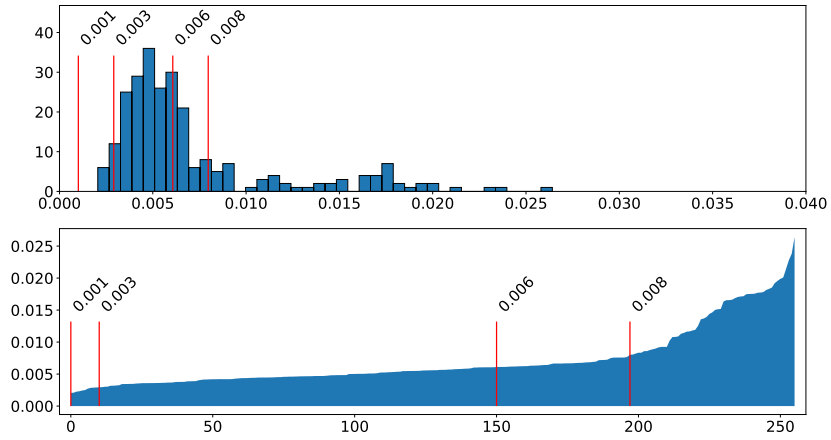

Figure 6: *Tf-Idf* analysis applied on LLVM `-O2` dataset. Distribution of the average *Tf-Idf* score measured per document (top). Average *Tf-Idf* score directly represented in a bar-plot. In the figure are depicted the score threshold used in *Tf-Idf* filtering evaluation.

Hyper-parameter	Init	Network	Training
Input size	-	2048	-
Padding/Truncating strategy	-	pre	-
Embedding size	-	128	-
Conv kernel size	-	9	-
Conv kernel number	-	32	-
Dense layer size	-	256	-
Batch size	32	-	64
Aux output weight loss	0.2	-	1.0

Table 3: Evolution of the CNN network and training hyper-parameters through the different grid-search phases. At first, we assumed batch-size equal to 32 and auxiliary output weight loss equal to 0.2. Then, we performed the network grid-search for establishing networks hyper-parameters. At last, we investigated batch-size and auxiliary output weight loss values with the training grid-search.

4.3. Hyperparameters exploration

The CNN architecture we propose for language modelling has a relatively high number of hyper-parameters, and the number of possible network configurations increases exponentially with the number of parameters considered. For this reason, we selected a set of hyper-parameters of interest and split them into two groups, optimised separately using two successive grid-searches. These groups are reported in Table 2. Notice that all networks are trained using the Adam optimiser with a starting learning rate of 0.001. Its additional hyper-parameters are set to the default values specified by Tensorflow. We leave to future works the evaluation of how optimisers impact the robustness of deep learning models for source code classification.

For selecting the best configuration among a set grid-search points, we adopted the following procedure:

1. Summarise each set of repetitions related to the same network configuration with its MCC mean and standard deviation.
2. Sort grid-search configurations by their average MCC.
3. Select the top 3 configurations.
4. Declare the configuration with the smallest standard deviation to be the best.

All experiments performed with Hyper-parameters exploration were made on the AMD dataset, using sequences in IR obtained trough a compilation without optimisation steps (`-O0`).

4.3.1. Exploration results

Figure 7 shows the performance in terms of accuracy against network hyper-parameters (top) and training hyper-parameters (bottom) configurations considered, sorted by mean accuracy. The first grid-search aims at optimising

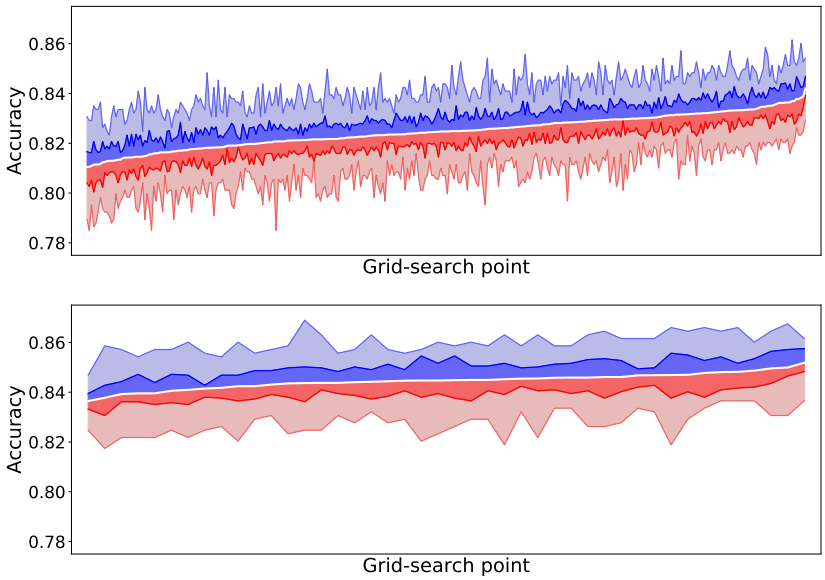

Figure 7: Classification accuracy improvements of network (top) and training (bottom) hyper-parameters exploration. The tracks represent classification performance quartiles of grid-search configurations explored, sorted by average accuracy. The mean trend is represented by the white track in the middle of each plot.

hyper-parameters related to the CNN network structure. The search space is the hyper-cube defined by the Cartesian product of the network parameters in Table 2, while batch-size and auxiliary output weight loss were fixed to 32 and 0.2, as specified by Table 3. The best configuration so far has a mean MCC of 0.672 and a standard deviation of 0.014. Mean classification accuracy equals 84.2% with a standard deviation of 0.7% and the best value of 86.2%.

The second grid-search improves the CNN performance selecting better batch-size and auxiliary output weight loss values, keeping the network hyper-parameters fixed. At the end of the two optimisation procedures, mean MCC approaches 0.693, with a standard deviation of 0.015, while classification accuracy reaches 85.2% with a standard deviation of 0.7% and the best value of 86.9%.

4.3.2. Parameters impact

We analysed the outcome of the grid-search to identify the hyper-parameters having more relevant impact on classification accuracy. Given an hyper-parameter p and two values A and B it assumes, we are interested in computing the distribution of the accuracy improvement granted by moving the value of p from A to B. For each CNN configuration C_A where p equals A, another one is selected, called C_B, characterised by having p equal to B and all the remaining hyper-parameters in common with C_A. Then, the classification accuracy difference is computed between C_A and C_B.

Figure 8 shows the outcome of such a procedure for all network and training

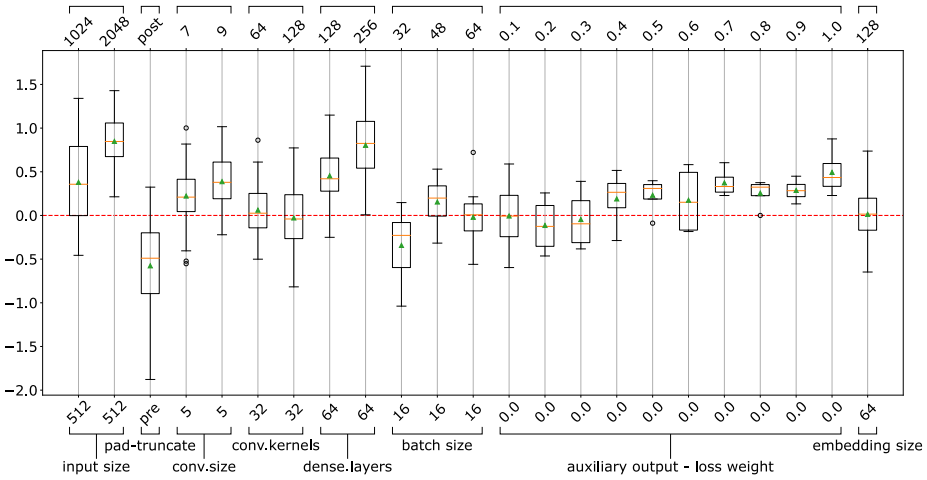

Figure 8: Single hyper-parameter sensitivity analysis in CNN models.

Hyper-parameter	RNN	CNN
Input size	1024	2048
Padding/Truncating strategy	pre	pre
Embedding size	128	64
Conv kernel size	-	9
Conv kernel number	-	32
LSTM layer 1 size	64	-
LSTM layer 2 size	64	-
Dense layer size	32	256
Batch size	32	64
Aux output weight loss	0.2	1.0

Table 4: Architecture of the two machine learning models tested.

hyper-parameters. It shows that sequence input-size and the number of neurons in the fully-connected layer of the classifier (dense-layers) are particularly promising. In essence, increasing them seems to be beneficial, and moving them from the minimum tested values to the maximum one always leads to better performance, no matter the values other hyper-parameters assume. Auxiliary output loss weight exposes a similar behaviour when exceeding 0.6. It is observed that increasing convolution kernel size from 5 to 9 generally leads to better results while applying padding to the end of the sequences is detrimental.

4.4. CNN-LSTM comparison

In this Section we compare CNN and LSTM models. We used the best model found using the grid search in Section 4.3 and we compared it with a network

Dataset	Filtering	Length	RNN			CNN		
			ACC [%]	MCC {-1, +1}	S.Up [x]	ACC [%]	MCC {-1, +1}	S.Up [x]
LLVM `-O0`	-	3092	81.16	0.614	1.37	84.47	0.685	**1.69**
	Blacklist	900	82.22	0.636	1.40	85.40	0.704	1.58
	1 e-3	651	81.90	0.630	1.33	**85.60**	**0.708**	1.61
	3 e-3	553	81.16	0.615	1.08	84.79	0.691	1.45
	6 e-3	497	81.45	0.620	1.33	83.59	0.666	1.47
	8 e-3	465	79.24	0.576	1.03	81.78	0.629	0.97
LLVM `-O2`	-	3160	80.73	0.606	1.42	83.77	0.670	**1.60**
	Blacklist	949	82.53	0.643	1.41	84.20	0.681	1.50
	1 e-3	719	**82.60**	**0.644**	1.38	**84.78**	**0.691**	1.57
	3 e-3	602	82.39	0.640	1.03	84.63	0.688	1.54
	6 e-3	515	81.64	0.624	0.94	83.38	0.661	1.57
	8 e-3	495	80.60	0.604	0.77	83.31	0.660	1.51
OpenCL	-	2656	81.18	0.615	**1.49**	**83.00**	**0.656**	**1.51**

Table 5: Outcomes of experiments comparing different language modelling networks and token filtering strategies on the Nvidia data-set in terms of classification accuracy and MCC. Also, we reported speedup with respect to a static kernel mapper, mapping each kernel on GPU.

Dataset	Filtering	Length	RNN			CNN		
			ACC [%]	MCC {-1, +1}	S.Up [x]	ACC [%]	MCC {-1, +1}	S.Up [x]
LLVM `-O0`	-	3092	79.83	0.581	3.23	**85.32**	**0.695**	3.50
	Blacklist	900	81.97	0.625	3.23	83.97	0.667	3.50
	1 e-3	651	82.08	0.627	3.21	84.79	0.684	**3.88**
	3 e-3	553	81.47	0.615	3.17	84.76	0.683	3.80
	6 e-3	497	80.55	0.596	3.38	83.74	0.662	3.48
	8 e-3	465	79.55	0.575	2.16	83.76	0.662	2.15
LLVM `-O2`	-	3160	79.25	0.568	3.01	84.54	0.679	3.86
	Blacklist	949	81.82	0.622	2.97	83.81	0.663	3.59
	1 e-3	719	81.50	0.615	3.34	**84.78**	**0.683**	**3.91**
	3 e-3	602	**82.50**	**0.637**	3.20	84.23	0.672	3.33
	6 e-3	515	80.35	0.592	3.08	83.52	0.657	3.24
	8 e-3	495	79.56	0.575	2.07	83.96	0.666	3.63
OpenCL	-	2656	81.75	0.622	**3.69**	**84.78**	**0.684**	**3.26**

Table 6: Outcomes of experiments comparing different language modelling networks and token filtering strategies on the AMD data-set in terms of classification accuracy and MCC. Also, we reported speedup with respect to a static kernel mapper, mapping each kernel on CPU.

exploiting an LSTM-based language modelling. Table 4 details the architecture of the two networks.

Tables 5 and 6 report the outcome of experiments performed on both the AMD and the Nvidia datasets. For each kernel available, we applied the processing procedures detailed in Section 3.1. Each kernel is tokenised and atomised using:

- the LLVM-based pipeline described in this work with two different *clang* optimisation flags, for investigating how source code transformation impacts on classification.
- the OpenCL-based approach proposed in [3].

Regarding LLVM token sequences, we tested two token filtering strategies:

- the blacklist approach proposed in [5].
- four different *Tf-Idf* thresholds.

We evaluated experiments in terms of classification accuracy, MCC and speedup. Red bold values highlight what preprocessing methodology gives the best result for each metric-model pair. Instead, the highlighted values stress the filtering strategies with which the CNN behaves better for each tokenising methodology.

The next two subsections discuss the classification performance of tested models and how CNN language modelling impact training time.

4.4.1. Kernel classification performance

The CNN model always outperforms the RNN one, independently from the device used for kernel labelling, the preprocessing strategies, the filtering threshold.

Concerning classification metrics, the CNN performs better on LLVM `-O0` sequences in the AMD dataset, where it reaches a mean classification accuracy of 85.32% and a MCC of 0.695. The CNN grants a boost in classification performance between +3.00% and +5.50% on unfiltered sequences, providing a solid improvement over RNN. The same conclusion can be drawn from Nvidia results which are characterised by a slight reduction of the gap between classification accuracy and MCC offered by the two models. Here, the best mean classification metrics achieved by the CNN are 84.47% accuracy and 0.685 MCC on LLVM `-O0` sequences.

Token filtering improves models classification performance of LLVM sequences in most cases. Token blacklist, proposed in [5], and *Tf-Idf* filtering with thresholds 0.001 and 0.003 are the three strategies that often behave better. RNN performances are more influenced by token filtering than the CNN ones. That is especially true in the AMD dataset, where using a threshold equal to 0.003 on the LLVM `-O2` sequences provides a boost of +3.25% in accuracy and +0.069 in MCC. Instead, the CNN is less sensitive to filtering strategies as the best improvement is +1.13%, achieved applying *Tf-Idf* filtering on LLVM `-O0` sequences. High *Tf-Idf* thresholds, such as 0.008, usually lead to bad performance,

Batch size	Input size									
	512		1024		2048		4096		8192	
	CNN	RNN	CNN	RNN	CNN	RNN	CNN	RNN	CNN	RNN
16	291	1832	289	2871	397	4630	510	8560	812	15995
32	173	1034	211	1647	302	3032	421	5305	797	10637
64	144	670	179	989	263	1722	488	3331	903	6066
128	106	430	159	633	289	1212	581	2141	996	4533

Table 7: Amount of time (in seconds) required for training CNN and RNN models for different batch-size and input-size. Experiments were run on an Nvidia Quadro RTX 6000 GPU with 24GB RAM.

Dataset	*DeepTune* [3]	*DeepLLVM*	
		LLVM `-O0`	LLVM `-O2`
AMD	3.43	3.50	3.86
Nvidia	1.42	1.69	1.60

Table 8: Comparisons between the speedup obtained by *DeepTune* [3] and the one obtained by the best CNN on IR.

causing an accuracy drop of 2.69% in the worst case. In those situations, the filtering algorithm starts removing tokens with high informative content not homogeneously, making to learn the appropriate features for carrying on the classification tasks challenging.

4.4.2. Training time comparison

Training time is one of the most significant issues when dealing with deep machine learning models. Since input-size and batch-size are the two hyper-parameters that affect training time most, comparing the two proposed models as they are, would not have been fair. We computed the average time CNN and RNN require for being cross-validated using sequences of the same length and an equal number of samples in mini-batches. For this analysis we also considered pairs of input size/batch size not considered during hyper-parameters optimisation, in order to provide a more complete set of results. As shown in Table 7, the CNN always performs consistently better than the RNN. In the worst case, it provides a speedup of 3.7x, if we consider the configuration with input size 4096 and batch size 128. At the opposite, testing the two networks with input size 8192 and batch size 16 gives the best possible speedup, equal to 19.7x.

4.5. Speedup comparison

For devising how the proposed machine learning model behaves in a real-world scenario, evaluating classification accuracy and MCC is not sufficient. Typical machine learning metrics take care of checking the number of correct predictions over the total number of samples but do not consider the impact of

mapping a kernel on the wrong device. Missing the best compute unit may not result in a significant penalty in terms of runtime or speedup.

For checking the impact of misclassified kernels, we measured the speedup granted by the two machine learning models tested. We computed Speedup using the same approach proposed in [3]. The time required for running all the OpenCL kernels on the device predicted by a classifier is divided by the time required for mapping all kernels on the device whose label is most common in the labelled dataset. Experiment results are summarised in Tables 5 and 6. The CNN always outperforms the RNN model on IR sequences analysis. It reaches a mean speedup of 1.69x and 1.60x on the Nvidia dataset while reaching 3.50x and 3.86x on AMD labelled data. Moreover, the best average speedup obtained by the CNN outperforms results presented in [3] on both dataset as shown in Table 8. The RNN module obtains better results analysing OpenCL kernels from the AMD dataset, ensuring a speedup of 3.69x. Nonetheless, this speedup is smaller than the 3.91x ensured by the CNN on LLVM `-O2` sequences filtered with a *Tf-Idf* threshold of 1e-3.

4.6. Additional comparisons and dataset considerations

While in the previous sections we focused on CNN vs LSTM comparison and hyper-parameters impact, in this Section we compare our method with other state-of-art papers using a different approach for source code analysis, but still using deep learning models.

In particular, we consider recent works [8, 10], where methodology using graph neural networks have been presented. For comparison, we adopted the same cross-validation approach used in all state-of-art papers, and we performed experiments using the validation method introduced in [8].

4.6.1. Preliminary considerations on validation method and literature results

Regarding validation, [8] suggests that the performance of a model can be evaluated on a collection of N benchmark suites running a N-fold cross-validation where each fold corresponds to a particular benchmark suite in the dataset. It ensures that a network is tested only on samples coming from a benchmark-suite not considered during training. Such procedure is called *grouped-split* cross-validation, whereas the procedure currently used is called *random-split*. Furthermore, in [8] are reproduced the results shown in previous works. The results proposed for the *random-split* methodology seem to overestimate the results previously obtained in the literature (e.g. *DeepTune* classification accuracy at 91% instead of 81%).

To perform the experiments, we reviewed the code attached to the work and identified a criticality[3] in the construction of the cross-validation procedure that once corrected reported the accuracy values coherent with the one available

[3]The fixed seed used during the creation of the folds (204) leads to obtain exceptional good accuracy values randomly from the cross-validation procedure. Furthermore, the same fold configuration is always evaluated during the replicas of the cross-validation procedure.

Fold	Test benchmark	Samples		Imbalance Train Set		Imbalance Test Set	
		Train	Test	AMD	NVD	AMD	NVD
0	AMD	664	16	+0.17	-0.12	+0.25	-0.88
1	NPB	153	527	-0.02	+0.22	+0.23	-0.24
2	NVidia	668	12	+0.19	-0.14	-0.83	-0.16
3	Parboil	661	19	+0.18	-0.15	0.0	+0.33
4	Polybench	653	27	+0.22	-0.14	-0.85	-0.11
5	Rodinia	649	31	+0.18	-0.16	+0.10	+0.22
6	SHOC	632	48	+0.15	-0.21	+0.46	+0.83

Table 9: Overview of fold statistics for *grouped-split* cross-validation. The last four columns report label imbalance of train and test sets for the AMD (AMD) and Nvidia (NVD) datasets computed as $(n_{\text{cpu}}-n_{\text{gpu}})/(n_{\text{cpu}}+n_{\text{gpu}})$. A positive ratio implies a bigger number of CPU labelled samples, while a negative ratio reports an higher number of GPU labelled samples.

in the literature. All results of [8] exposed in the *random-split* procedure are affected by this problem[4].

Concerning the group-split method, all results of [8] highlight a significant reduction in classification performance of any methodologies proposed in literature so far.

We claim that the while methodology proposed in [8] is promising, the *grouped-split* method in the considered dataset may lead to inconsistent results. Indeed, using benchmarks suites in place of folds during cross-validation leads to a number of pitfalls. (i) The methodology cannot be defined *stratified cross-validation* since each benchmark has its own class imbalance. (ii) Cross-validation wishes for uniformly sized folds. As shown in Table 9 this is not the case for the used dataset. As a result, the evaluation of classification performance is likely to be biased (e.g. NPB benchmark). (iii) In the considered dataset each fold is affected by high variability in class imbalance. Such a phenomenon is quite evident in both datasets (e.g. AMD dataset - NVidia and Polybench benchmarks, Nvidia dataset - AMD and SHOC benchmarks).

4.6.2. Results comparison

Even if we question the usage of *grouped-split* cross-validation methodology on the available dataset, we tested the proposed CNN model as detailed in [8] for the sake of completeness. Results are presented in Table 10, where we report the results related to random-split, both the numbers reported in [8] (called fixed-seed) as well as the one we obtained by fixing the seed problem (random

[4]By eliminating the use of fixed seed, *DeepTune* accuracy has returned from 91% to 81% (five repetitions) in the range of values previously described in [3, 5]. Even if the same seed is used for the evaluations done in [3], the use of different versions of the libraries leads to different fold configurations. We report that we were not able to replicate results for GNN-CDFG because of problems in the provided artefacts, as such we report the results shown in [8].

Method	*DeepTune* [3]	*ProGraML* [10]	GNN-CDFG [8]	*DeepLLVM*
random-split (fixed seed) [3]	92%	-	93%	-
random-split (random seed) [4]	**81%**	**83%**	**-**	**85%**
grouped-split	**48**%	**-**	**52%**	**53%**

Table 10: Summary of IR based source code analysis for heterogeneous device mapping. The table reports the mean classification accuracy between AMD and Nvidia datasets. *Grouped-split* results for *ProGraML* were not provided in [10].

seed). Missing numbers in Table 10 refers to cases for which literature numbers are not present, or the experiments could not be replicated.

Overall, by considering the random-split (random seed) and grouped-split results, our method implemented with DeepLLVM performs better than other state-of-art in terms of accuracy, by keeping the advantages of a CNN model in terms of training time. Also considering the speedup, the proposed method outperforms the results reported in [8] that achieves 1.61 for the AMD dataset, compared to 3.50 and 3.86 as reported in Table 8.

4.7. Summary of findings

The results obtained highlight the following main findings:

- Using CNN-based language modelling networks, in the context of kernel-device mapping, is a promising method for extracting features from source code. They provide mean classification accuracy, MCC and speedup higher than the one provided by the LSTM-based network, reaching 85.32% classification accuracy, 0.695 MCC and 3.86x speedup in the best cases for the considered dataset;
- With respect to LSTM models, CNN is more robust with respect to token filtering techniques, that seem to be less effective on convolutive networks;
- CNN architecture we proposed requires much lower GPU training time with respect to the LSTM one. Indeed, CNN ensures a 4x - 7x reduction in training time over RNN. It means that such models lend their-self to be explored in a more effective way, allowing to test different architectures and hyper-parameters configurations extensively;
- The study of the impact of CNN hyper-parameter on the classification metrics shows that most relevant ones are: Input sequences size, number of units in the fully-connected layer of the classifier, auxiliary output weight loss.
- The LSTM network reports a more significant variation in performance when the sequence length is reduced. At the same time, CNN is less affected by *Tf-Idf* filtering because of the Global Max Pooling layer acts as a filter itself.

- The proposed CNN-based method implemented in *DeepLLVM* outperforms state-of-art approaches and demonstrated to be robust w.r.t. hyper-parameter setting.

5. Conclusion

In this work, we presented a LLVM based code classification method, and we explored the impact of neural network models on feature extraction and classification problems applied to source code in the intermediate representation.

Given the absence, in literature, of a CNN-based language modelling network for kernel-device mapping, we explored the hyper-parameters of a CNN model in order to obtain a reference model. We explored 368 different hyper-parameters configuration, each cross-validated 20 times, reporting a statistical analysis of the results obtained.

We compared the best configuration of hyper-parameters for the CNN with the RNN-based network used in [3, 5] for different source code preprocessing and token filtering strategies, evaluating classification accuracy, MCC and speedup.

Results confirm that features extraction from IR is a valuable strategy for analysing sources without dealing with complex high-level constructs, and it can be done keeping all the information required for performing classification tasks in the context kernel-device mapping.

The use of SOTA natural language processing models such as GPT-3 will be considered in future work to determine the complexity of the OpenCL and LLVM-IR code classification task. It is currently an open issue to evaluate the performance of these models in a few-shot mode for domain-specific problems such as the one we face in our work. In this context, however, we should consider the size of the networks used for domain-specific tasks. In our work, the CNN network consisted of $3.4 * 10^4$ parameters and the LSTM network of $9.8 * 10^4$ parameters about seven orders of magnitude smaller than GPT-3, which counts about $1.7 * 10^{11}$ parameters.

We will explore other fields of application of the methodologies proposed in this work. For example, in many cases, a source code fragment under analysis is highly dependent on a specific software library, this is the case of Spiking Neural Networks (SNN) simulations, described through a high-level software library. Several works in the literature have provided a formalism to deal with the problem of SNN placement on an MCSoC architecture (SpiNNaker, Loihi) describing both the SNN and the architecture through a graph [36–38]. The possibility to analyse such problem training a model able to infer the optimal placement (or learn features to simplify the mapping procedure) is, therefore, an open problem on which we want to investigate.

Acknowledgments

This work was supported in part by the Italian Ministry for Education, University and Research (MIUR) under the program "Dipartimenti di Eccellenza"

(2018-2022). G.U. work has been funded by ECSEL Joint Undertaking (EU H2020) under the Arrowhead Tools research project with Grant Agreement no. 826452.

References

[1] Z. Wang, M. O'Boyle, Machine learning in compiler optimization, Proceedings of the IEEE 106 (11) (2018) 1879–1901.

[2] G. Carvalho, B. Cabral, V. Pereira, J. Bernardino, Computation offloading in edge computing environments using artificial intelligence techniques, Engineering Applications of Artificial Intelligence 95 (2020) 103840. doi:https://doi.org/10.1016/j.engappai.2020.103840.
URL http://www.sciencedirect.com/science/article/pii/S0952197620302050

[3] C. Cummins, P. Petoumenos, Z. Wang, H. Leather, End-to-end deep learning of optimization heuristics, in: Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on, IEEE, 2017, pp. 219–232.

[4] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society, 2004, p. 75.

[5] F. Barchi, G. Urgese, E. Macii, A. Acquaviva, Code mapping in heterogeneous platforms using deep learning and llvm-ir, in: 2019 56th ACM/IEEE Design Automation Conference (DAC), IEEE, 2019, pp. 1–6.

[6] T. Ben-Nun, A. S. Jakobovits, T. Hoefler, Neural code comprehension: a learnable representation of code semantics, in: Advances in Neural Information Processing Systems, 2018, pp. 3585–3597.

[7] V. K. S, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, Y. N. Srikant, Ir2vec: A flow analysis based scalable infrastructure for program encodings (2019). arXiv:1909.06228.

[8] A. Brauckmann, A. Goens, S. Ertel, J. Castrillon, Compiler-based graph representations for deep learning models of code, in: Proceedings of the 29th International Conference on Compiler Construction, CC 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 201–211. doi:10.1145/3377555.3377894.
URL https://doi.org/10.1145/3377555.3377894

[9] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, On the feasibility of transfer-learning code smells using deep learning, arXiv preprint arXiv:1904.03031 (2019).

[10] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, H. Leather, Programl: Graph-based deep learning for program optimization and analysis (2020). arXiv:2003.10536.

[11] J.-W. Son, T.-G. Noh, H.-J. Song, S.-B. Park, An application for plagiarized source code detection based on a parse tree kernel, Engineering Applications of Artificial Intelligence 26 (8) (2013) 1911 – 1918. doi:https://doi.org/10.1016/j.engappai.2013.06.007.
URL http://www.sciencedirect.com/science/article/pii/S0952197613001085

[12] M. Allamanis, E. T. Barr, P. Devanbu, C. Sutton, A survey of machine learning for big code and naturalness, ACM Computing Surveys (CSUR) 51 (4) (2018) 1–37.

[13] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, C. Silvano, A survey on compiler autotuning using machine learning, arXiv preprint arXiv:1801.04405 (2018).

[14] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[15] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, Q. V. Le, Xlnet: Generalized autoregressive pretraining for language understanding, in: Advances in neural information processing systems, 2019, pp. 5753–5763.

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners (2020). arXiv:2005.14165.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.

[18] A. Monsifrot, F. Bodin, R. Quiniou, A machine learning approach to automatic production of compiler heuristics, in: International conference on artificial intelligence: methodology, systems, and applications, Springer, 2002, pp. 41–50.

[19] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, Y. Gao, Exploiting statistical correlations for proactive prediction of program behaviors, in: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, ACM, 2010, pp. 248–256.

[20] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, O. Temam, Rapidly selecting good compiler optimizations using performance counters, in: Code Generation and Optimization, 2007. CGO'07. International Symposium on, IEEE, 2007, pp. 185–197.

[21] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, C. Silvano, Cobayn: Compiler autotuning framework using bayesian networks, ACM Transactions on Architecture and Code Optimization (TACO) 13 (2) (2016) 21.

[22] M. Stephenson, S. Amarasinghe, Predicting unroll factors using supervised classification, in: Proceedings of the international symposium on Code generation and optimization, IEEE Computer Society, 2005, pp. 123–134.

[23] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, P. Sadayappan, Predictive modeling in a polyhedral optimization space, International journal of parallel programming 41 (5) (2013) 704–750.

[24] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, S. Amarasinghe, Autotuning algorithmic choice for input sensitivity, in: ACM SIGPLAN Notices, Vol. 50, ACM, 2015, pp. 379–390.

[25] E. Park, J. Cavazos, M. A. Alvarez, Using graph-based program characterization for predictive modeling, in: Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, 2012, pp. 196–206.

[26] D. Grewe, Z. Wang, M. F. O'Boyle, Portable mapping of data parallel programs to opencl for heterogeneous systems, in: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2013, pp. 1–10.

[27] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, nature 323 (6088) (1986) 533–536.

[28] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[29] Y. LeCun, et al., Generalization and network design strategies, Connectionism in perspective 19 (1989) 143–155.

[30] W. Yin, K. Kann, M. Yu, H. Schütze, Comparative study of cnn and rnn for natural language processing, arXiv preprint arXiv:1702.01923 (2017).

[31] G. Urgese, L. Peres, F. Barchi, E. Macii, A. Acquaviva, Work-in-progress: Multiple alignment of packet sequences for efficient communication in a many-core neuromorphic system, in: 2018 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), IEEE, 2018, pp. 1–2.

[32] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, C. Potts, Learning word vectors for sentiment analysis, in: Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1, Association for Computational Linguistics, 2011, pp. 142–150.

[33] E. Kaufmann, A. Bernstein, L. Fischer, Nlp-reduce: A naive but domain-independent natural language interface for querying ontologies, in: 4th European Semantic Web Conference ESWC, 2007, pp. 1–2.

[34] Y. Zhang, B. Wallace, A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification, arXiv preprint arXiv:1510.03820 (2015).

[35] G. Jurman, S. Riccadonna, C. Furlanello, A comparison of mcc and cen error measures in multi-class prediction, PloS one 7 (8) (2012) e41882.

[36] G. Urgese, F. Barchi, E. Macii, Top-down profiling of application specific many-core neuromorphic platforms, in: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, IEEE, 2015, pp. 127–134.

[37] G. Urgese, F. Barchi, E. Macii, A. Acquaviva, Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms, IEEE Transactions on Emerging Topics in Computing 6 (3) (2016) 317–329.

[38] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, et al., Loihi: A neuromorphic manycore processor with on-chip learning, IEEE Micro 38 (1) (2018) 82–99.