# Deep Learning and Classical Machine Learning for code mapping in Heterogeneous Platforms

Yacine Hakimi
*Laboratoire de Méthodes de Conception des Systèmes (LMCS)*
*Ecole nationale Superieure d'Informatique (ESI)*
Algiers, Algeria
y_hakimi@esi.dz

Riyad Baghdadi
*Computer Science & Artificial Intelligence Lab (CSAIL)*
*Massachusetts Institute of Technology (MIT)*
de Massachusetts, USA
baghdadi@mit.edu

Yacine Challal
*Laboratoire de Méthodes de Conception des Systèmes (LMCS)*
*Ecole nationale Superieure d'Informatique (ESI)*
Algiers, Algeria
y_challal@esi.dz

***Abstract*—Programming modern heterogeneous systems is becoming more and more challenging due to their complexity. To simplify software development for such architectures, more advanced compilers are being designed. Such compilers automatically optimize code and hide the complexity of the target heterogeneous architecture from the developer. An example of problems that these compilers need to solve is to decide whether to map (run) a piece of code on CPU or GPU (Graphics Processing Unit). State-of-the-art compilers use accurate optimization heuristics to solve such problems and decide about how to optimize a code automatically. Traditional machine learning and Deep Learning approaches have both been used to build such heuristics. While traditional machine learning is well suited for training on small datasets, it is not well suited for extracting a set of high-quality features, on the other hand, deep learning can automatically extract features, but it needs a large amount of data to get satisfactory results.**

**In this paper, we propose a new machine-learning-based model that allows a compiler to automatically decide whether to map a piece of code to CPU or GPU. Our model uses traditional machine learning and deep learning by exploiting the advantages of each of them. We show that our proposed model, which requires a small amount of data and training time, matches and outperforms state-of-the-art pre-trained deep learning models that require large amounts of data.**

***Index Terms*—Machine Learning, Deep Learning, Code Optimizations, LLVM-IR, Heterogeneous Platforms.**

## I. Introduction

When we try to write a parallel code or parallelize a legacy code there are a lot of options that we have to choose from, from selecting the algorithm and parallel programming model to determining the granularity of parallelism, scheduling, and mapping strategy, therefore there are many tunable parameters that can affect a parallel execution of the program. Ideally, optimal tuning of all these parameters can help in obtaining efficient execution, and in improving the performance. We can look at this problem as a problem of optimization, where we have to select the optimal or near-optimal solution from a set of possible solutions considering different criteria.

Writing parallel code becomes more challenging when we target heterogeneous systems which have different types of hardware accelerators (CPUs, GPUs - Graphics Processing Units-, etc.). Optimizing compilers are used to simplify writing software for such architectures. These compilers automatically decide about how the code should be optimized for the target architectures. One decision that such compilers need to take is to figure out whether to map a piece of code (kernel) to CPU or GPU [1] [2] [3] [4].

Because of the great successes in recent years in the field of machine learning, especially deep learning, the tendency is to use it in many compiler and code optimization problems [5] [6]. In previous work, deep learning has shown its ability to provide better accuracy and solve the problem of feature extraction and selection [1], but it requires large amounts of labeled data, which is a real and well-known problem in compiler and code optimization (because of lack of data). On the other hand, simple models such as decision trees and SVM typically work better and require less labeled data compared to deep learning models but their performance is greatly affected by the quality of features. Generally, they work well with a feature vector that has a small number of dimensions. Extracting a small number of high-quality features is a nontrivial task.

While feature extraction in traditional machine learning algorithms depends on the developer's experience and similar previous experiences [5], automated feature extraction using deep learning for code has been greatly influenced by Natural Language Processing (NLP) approaches [1] [7] (due to similarities between code and natural language). These approaches especially pre-trained models such as CodeBERT has given good results in code tasks similar to NLP tasks (code completion, code to code translation, code summarization…etc.) [8]. However, compiler and code optimization tasks are more difficult due to the structural nature of the code (function calls, branching) and the complexity of interactions between

978-1-6654-2821-7/21/$31.00 

programs and architectures, and the fact that features extracted have to characterize the semantics of the program and its behavior on the target architecture.

To overcome this problem many previous papers attempt not only to learn a distributed representation of the code in continuous space using embedding techniques but add other information to this representation such as control and data flow [7] [4] [9].

We believe that in NLP and code similar-NLP tasks that the key to their success is the similarity between the tasks that are used to learn the distributed representation in pretrained models and the downstream tasks because most of these techniques rely on masked language modeling where some tokens are masked and the model tries to predict them, while in code optimization the tasks are far from this type of task.

In this paper, we train a language model that can extract a high-quality features vector with a small number of dimensions relying on Convolutional Neural Networks CNNs. The characteristics of the features vector allow us to use traditional machine learning techniques which in turn allows us to use a small dataset.

We show that even with a very small dataset we can take advantage of deep learning and traditional machine learning together to reach results similar or better than state-of-the-art pre-trained models that require a large dataset. Unlike such models, our model requires less training time and less data.

The contributions of this paper can be summarized as follows:

- We propose a new model that maps kernels to CPU or GPU;
- The proposed model is the first to match the performance of state-of-the-art pre-trained models without requiring large amounts of data.

## II. RELATED WORK

In the last decade, there have been many attempts to use machine learning in compiler and code optimization such as scheduling kernels on CPU/GPU heterogeneous platforms using SVM [2], Decision trees [3] selecting the best value for the coarsening factor on GPU using ANN [10], determine whether parallelism is beneficial using SVM [11], and many other problems. A comprehensive survey can be found in [12].

The performance of these traditional machine learning techniques is greatly affected by the quality of features. Extracting high quality features requires expert intervention and a lot of time, and these features are not always useful for solving other optimization problems even for the same code [5].

Due to the common characteristics between source code and natural languages and the huge success of deep learning in the area of NLP, researchers have turned to use these techniques for compilers and code optimization to solve the problem of features extraction.

Working on the problem of device mapping on heterogeneous architectures, Cummins et al. [1] try to automatically extract features from OpenCL source code using embedding techniques to learn a distributed representation of the code. Their model, DeepTune, succeeded in obtaining better results than manual feature extraction methods.

Using the intermediate representation (IR) of the LLVM compiler (LLVM IR) instead of OpenCL code and working on the same problem, Barchi et al. [13] try to take advantage of the features of the LLVM IR to improve the accuracy. Inst2vec [7] uses the skip-gram model to train an embedding layer analyzing the ConteXtual Flow Graph (XFG) to add more information to the distributed representation, this pre-trained model has shown good results in many downstream tasks including device mapping problems.

Relying on Graph Neural Networks (GNN) and working on the same problem, Cummins et al. [4] proposed a graph-based program representation called Program Graphs for Machine Learning (PROGRAML), In the same direction, Brauckmann et al. [9] try to use abstract syntax trees (ASTs) and control-data flow graphs (CDFGs) as a code representation instead of code sequence to do the classification.

In IR2VEC [14] the authors proposed for the first time the use of Knowledge Graph Embeddings (KGE) in code optimization to learn a code representation using TransE model. This representation was appropriate to be used as an input to an XGboost classifier, this work is considered the state-of-the-art (SOTA) in the device mapping problem.

Barchi et al. [15] introduced the use of Convolutional Neural Networks (CNNs) instead of Recurrent Neural Networks (RNNs). The work showed CNN's ability to get good results, even better than many models that relied on pre-trained techniques using the small dataset of the device mapping problem only.

In this work, we take advantage of deep learning (CNN's capability of learning and extracting good features) and traditional machine learning together in code optimization. Our work shows that even with a very small dataset that only has 680 data points and without a pre-trained model trained on a large dataset, we can with our approach outperform the SOTA models.

## III. PROPOSED APPROACH

In Our work we use like the recent works the intermediate representation IR-LLVM as an input to the system, Figure 1 shows all the steps needed to build our model. Our system is composed of three stages as shown in Figure 2, A. Preprocessing of the IR-LLVM code, B.Training a CNN model, C. Training a ML model using the feature extractor built based on the embedding and CNN layers from the CNN model.

### A. Preprocessing

We preprocessed the IR-LLVM code by doing the following in this order: (1) Tokenizing the code, (2) Atomizing the code, figure 3 shows an example of the preprocessing stage.

*1) Tokenization:* in this phase, we have taken a slightly different approach comparing to work in [15]. before identifying the elements or the tokens in the sequence of code we rewrite the code by:

- Removing the empty line, comments, and unnamed metadata starting with (#,);
- Replacing the vectors, arrays and constant (float, integer, double) by a placeholder respecting the types. (e.g. 0x3FD3333340000000 → float_constant, 23 → int_constant, <float 1.000000e+01, float 1.000000e+01>→ vector_float_constant);
- Replacing global and local unnamed identifiers, variable, and function names with a placeholder.

```
Input: IR-LLVM code dataset
Output: A predictive model for a device mapping problem
Def Build_Predictive_Model(dataset.code):

    // Define preprocessing function
    Def preprocessing (code):
        code = Tokinization (code)
        code = Atomization (code)
        return code

    //Build a CNN model
    Def CNN_model (input):
        x = Embedding(input)
        x= Conv1d(x)
        x = GlobalMaxpooling1D(x)
        Output = Dense (x)

    //Build a Feature extractor model based on CNN
    model's layers (Embedding and Conv1d)
    Def Feature_extractor_Model (input, auxiliary_inputs):
        x = CNN_model. Embedding(input)
        x = CNN_model. Conv1d(x)
        x= GlobalMaxpooling1D(x)
        Features_vector = Concatenate (x, auxiliary_inputs)

    //Build Our predictive model
    Def Our_predictive_Model(input):
        Features_vector = Feature_extractor_Model (input)
        Prediction = ML (Features_vector)

    // Preprocess each code in the training dataset
    For each code in Tdataset.code do:
        dataset.code = Preprocessing (code)

    //Train the CNN model on the training dataset
    Train (CNN_model(Tdataset.code), dataset.labels)

    //Extract features vector for each code in training dataset
    Dataset.features = Feature_extractor_Model
    ( dataset.code,  dataset.auxiliary_inputs)

    // Select a machine learning algorithm
    ML = Select_Machine _learning_ algorithm ()

    // Train the ML on the same
    Train (ML, Dataset.features, labels)

    Return Our_predictive_Model()
```

Fig. 1. Build_Predictive_Model: Algorithm showing all steps to build our model.

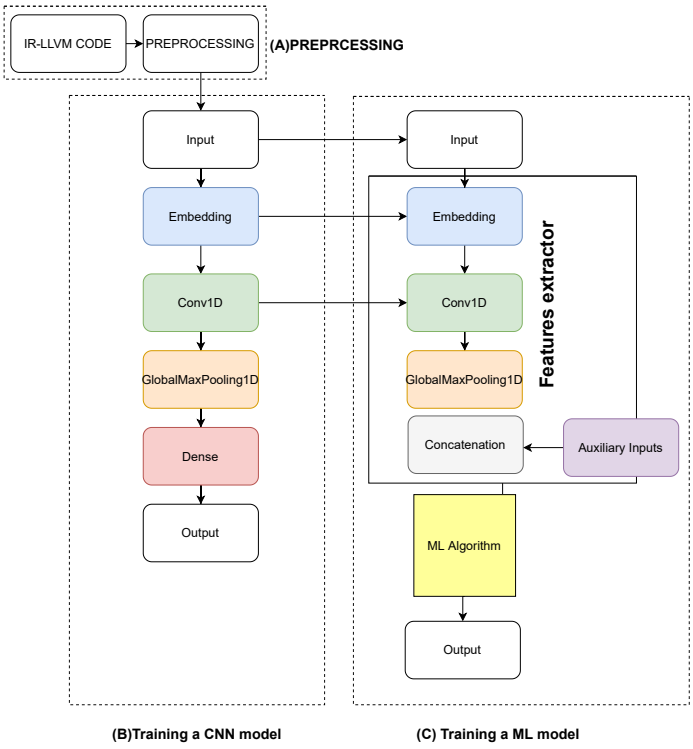


Fig. 2. System Overview: First training a CNN model, then building a Feature extractor model by transfer learning, and finally training a ML algorithm using the output of the Feature extractor as input.

After this process, we finally got a small vocabulary with 164 tokens unlike work in [15].

*2) Atomization:* this phase we created a dictionary to index tokens to integers, using this dictionary all tokens are converted to their integer indexes, finally we get a sequence of integers representing each code.

### *B. Training a CNN model*

Figure 2 (B) shows the architecture of our CNN model used in this stage, we use the same architecture used for the language model in [15].

*1) CNN model architecture:* this model consists of 4 layers:

*a) Embedding layer:* take a sequence of integers as input and learn a useful distributed representation of it by translating each integer (index of token) into a vector in the distributed representation. We choose an input size = 4096 and we use an Embedding size = 64.

*b) One-dimensional convolutional neural network:* take the distributed representation as input and applies 1D convolution over the input using a number of filters, each filter has a kernel size that defines the number of tokens to consider as the convolution is passed across the input, we use 64 filters with kernel size = 32.

*c) GlobalMaxpooling1D:* this layer selects the maximum value of each output of filters, the output is a vector with a size of 64.

*d) Dense:* a fully connected layer with 2 outputs and ”Sigmoid” activation to do the classification.

IR-LLVM CODE

```
%20 = fdiv float 1.000000e+00, %19, !fpmath !12
%21 = insertelement <4 x float> undef, float %20, i32 0
%22 = shufflevector <4 x float> %21, <4 x float> undef, <4 x i32> zeroinitializer
%23 = fmul <4 x float> %22, %18
%24 = tail call <4 x float> @_Z4sqrtDv4_f(<4 x float> %23) #4
%25 = fmul <4 x float> %24, <float 0x3FD3333340000000, float 0x3FD3333340000000, float 0x3FD3333340000000, float 0x3FD3333340000000>
```

(1) Tokenization

```
_local  =  fdiv float _float_constant  ,  _local
_local  =  insertelement 4_float undef ,  float _local ,  i32 _int_constant
_local  =  shufflevector 4_float _local ,  4_float undef ,  4_i32 zeroinitializer
_local  =  fmul 4_float _local ,  _local
_local  =  tail call 4_float _function ( 4_float _local )
_local  =  fmul 4_float _local ,    _vectorC_float
```

(2) Atomization

```
1, 5, 54, 11, 16, 2, 1,
1, 5, 85, 62, 74, 2, 11, 1, 2, 21, 4,
1, 5, 113, 62, 1, 2, 62, 74, 2, 82, 112,
1, 5, 25, 62, 1, 2, 1,
1, 5, 23, 20, 62, 19, 18, 62, 1, 17,
1, 5, 25, 62, 1, 2, 96
```

Fig. 3. Example of transformation of an intermediate representation to a sequence of numbers.

*2) Training :* The training was performed for 40 epochs using the Adam optimizer with its default parameters on the training dataset.

### C. Training a Machine Learning model

Figure 2 (C) shows the architecture of our classifier, we build another CNN model as a Feature extractor with the transfer learning technique, the output of this model concatenated with auxiliary inputs is used as inputs to train a ML algorithm.

*1) Feature extractor:* we built this model using the Embedding and the Conv1D layers from the pre-trained CNN model, this model consists of these two layers, a GlobalMaxpooling1D layer and a Concatenation layer used to add the Auxilliary inputs to the output. the result is a vector with 66 values (64 from the output of GlobalMaxpooling1D + 2 Auxilliary inputs). This model is not trained, it is only used as a feature extractor, it takes a sequence of integers and auxiliary inputs as input and produced Features vector as output.

*2) Auxiliary Inputs:* the auxiliary inputs added to feature vectors are real values used in the previous works to augment the source code input. The code optimization depends on many variables including the hardware, dynamic values can provide to the model important characteristics of the hardware, the auxiliary inputs used are workgroup size and byte transfer.

*3) Algorithms:* Once we get a features vector that has a small number of dimensions, we can train any Machine Learning algorithm to do the device mapping task and take advantage of its ability to deal with a small dataset. In Our work, we train 7 Machine learning algorithms and compared their results. Section IV shows the results of the comparison and discusses them.

*4) Ensemble Learning:* Ensemble Learning: ensemble learning is one of the promising techniques to deal with small and imbalanced datasets [16]. This technique uses a set of machine learning algorithms to do the classification and then uses their results to do the final classification. Since we can use traditional ML with our feature extractor, we can take advantage of this technique. In our work, we use StackingClassifier, in this model the output of each ML algorithm (classifier) in the ensemble is stacked and use a classifier (Meta-classifier) to compute the final prediction from this stack. Our ensemble learning is a set of Xgboost classifier and RandomForestClassifier as a classifier and AdaBoost Classifier as a meta-classifier.

## IV. EVALUATIONS

In this section, we evaluate our model on the device mapping problem using the same dataset and metrics used in previous works.

### A. Dataset

- In our work we use the dataset used in [14], this dataset is the same dataset used in the previous works on device mapping problem including [1], the only difference between these two is that in the first the OpenCL code was already converted to IR-LLVM code.
- The dataset is composed of 680 data for each of the two GPU devices (NVIDIA GTX 970 and AMD Tahiti 7970) obtained by executing 256 unique kernels on CPU and GPU with different workgroups size and byte transfer values. these kernels are from different benchmark suites comprising AMD SDK, NPB, NVIDIA SDK, Parboil, PolybenchGpu, Rodinia, and SHOC. each data is composed by kernel converted to IR-LLVM code, the optimal target device CPU or GPU, parameters (workgroup size, byte transfer).

TABLE I
COMPARISON OF ML ALGORITHMS ACCURACY USING THE FEATURE EXTRACTOR

| dataset | Algorithm | | | | | | |
|---|---|---|---|---|---|---|---|
| | ***XG boost*** | ***Random forest*** | ***Ada Boost*** | ***SVM*** | ***Gaussian NB*** | ***Logistic Regression*** | ***Ensemble Learning*** |
| AMD Tahiti | 90.58% | 89.53% | 89.83% | 70.85% | 64.12% | 40.95% | 92.22% |
| Nvidia | 88.19% | 82.81% | 85.20% | 58.59% | 53.36% | 57.24% | 91.03% |
| Average | 89.38% | 86.17% | 87.51% | 64.72% | 58.74% | 49.10% | 91.62% |

TABLE II
COMPARISON OF THE ACCURACY WITH THE STATE-OF-THE-ART MODELS

| dataset | Model | | | | | | |
|---|---|---|---|---|---|---|---|
| | ***Grewe et al.*** | ***DeepTune*** | ***Inst2vec (NCC)*** | ***ProGraML*** | ***DeepLLVM*** | ***IR2VEC*** | ***Our System*** |
| AMD Tahiti | 73.38% | 83.67% | 82.79% | 86.6% | 85.60% | 92.82% | 92.22% |
| Nvidia | 72.94% | 80.29% | 81.76% | 80.0% | 85.32% | 89.68% | 91.03% |
| Average | 73.16% | 81.98% | 82.27% | 83.3% | 85.46% | 91.25% | 91.62% |

### *B. Metrics*

Like previous works and to be able to compare with it, we use the accuracy metrics with 10-fold cross-validation to evaluate our system. The data is randomly split into 10 sets and we train 10 classifiers changing the subset used as test-set. We first train the CNN model on the train-set, then we build the Feature extractor model by transfer learning, and finally, we train the ML algorithm on the same train-set using the output of the Feature extractor as input. The prediction is done by the model shown in Figure 2 (C) on the test dataset to compute the accuracy.

### *C. Machine learning algorithms comparison*

We compare the accuracy of 7 ML algorithms including the Ensemble learning, Results are presented in Table 1.

The parameters we used in each model are as follows:

- Xgboost: max_depth = 6, learning_rate = 0.1, n_estimators = 100, n_jobs =10;
- Random forest: max_depth = 5, n_estimators = 50;
- AdaBoost, SVM, GuassianNB, LR: Default parameters;
- Ensemble Learning: The models used in it have the same parameters as the previous ones.

As Table 1 shows, the ensemble technique tree-based algorithms (Xgboost, Randomforest, Adaboost) provide good results in contrast to the rest, and this is to be expected since we are dealing with a very small dataset, also, the goal of extracting appropriate features for traditional machine learning was to be able to take advantage of these techniques. We build our ensemble learning based on these three algorithms and it provides the best accuracy.

### *D. Comparison with previous works*

We compare the accuracy of our system with the previous works, Table 2 shows this comparison. Our system provides roughly the same accuracy compared to the state-of-the-art model IR2VEC on the AMD dataset while providing better accuracy than it on the Nvidia dataset. Our model outperforms all other models despite we use only the small dataset of device mapping and without trying to add any other information such as control and data flow.

Our model provides similar or better results than state-of-the-art models with less data and without any additional information or pre-trained on large data.

## V. CONCLUSION AND FUTURE WORK

In this work we proposed a new model that maps kernels to CPU or GPU, our model provides similar or even better performance compared to state-of-the-art models. Our approach match and even outperforms the pre-trained models used only a very small dataset by using the advantages of deep learning and traditional machine learning together.

We used a deep learning CNN based model to build a Features extractor by transfer learning, this Features extractor model was able to extract features suitable for use by traditional machine learning algorithms, which allowed us to take advantage of known techniques to deal with the problem of small dataset such as ensemble learning.

We believe that our model can be used on other code optimization problems since the lack of labeled data is a well-known problem in this field.

Evaluate the performance of this model on other code optimization problems or even on the similar-NLP code classification tasks will be considered in future work. Another direction could be to try to improve the Features extractor to get better features and thus better performance.

## REFERENCES


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

[2] Y. Wen, Z. Wang, and M. F. O'boyle, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms," in *2014 21st International conference on high performance computing (HiPC)*. IEEE, 2014, pp. 1–10.

[3] D. Grewe, Z. Wang, and 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.

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

[5] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[6] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[7] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *arXiv preprint arXiv:1806.07336*, 2018.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[9] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 201–211.

[10] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 455–466.

[11] Z. Wang, G. Tournavitis, B. Franke, and M. F. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–26, 2014.

[12] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic, "Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review," *Computing*, vol. 101, no. 8, pp. 893–936, 2019.

[13] F. Barchi, G. Urgese, E. Macii, and 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.

[14] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.

[15] F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva, "Exploration of convolutional neural network models for source code classification," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104075, 2021.

[16] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, no. 2, pp. 241–258, 2020.