CV4Code: Sourcecode Understanding via Visual Code Representations

Ruibo Shi, Lili Tao, Rohan Saphal, Fran Silavong, Sean Moran

CTO, JP Morgan Chase

Abstract. We present CV4Code[1], a compact and effective *computer vision* method for sourcecode understanding. Our method leverages the contextual and the structural information available from the code snippet by treating each snippet as a two-dimensional image, which naturally encodes the context and retains the underlying structural information through an explicit spatial representation. To codify snippets as images, we propose an ASCII codepoint-based image representation that facilitates fast generation of sourcecode images and eliminates redundancy in the encoding that would arise from an RGB pixel representation. Furthermore, as sourcecode is treated as images, neither lexical analysis (tokenisation) nor syntax tree parsing is required, which makes the proposed method agnostic to any particular programming language and lightweight from the application pipeline point of view. CV4Code can even featurise syntactically incorrect code which is not possible from methods that depend on the Abstract Syntax Tree (AST). We demonstrate the effectiveness of CV4Code by learning Convolutional and Transformer networks to predict the functional task, *i.e.* the problem it solves, of the source code directly from its two-dimensional representation, and using an embedding from its latent space to derive a similarity score of two code snippets in a retrieval setup. Experimental results show that our approach achieves state-of-the-art performance in comparison to other methods with the same task and data configurations. For the first time we show the benefits of treating sourcecode understanding as a form of image processing task.

Keywords: Sourcecode Understanding, ResNet, Transformer, ViT

1 Introduction

Machine Learning on Sourcecode (MLOnCode) promises to redefine how software is delivered through intelligent augmentation of the software development lifecycle (SDLC). Automation of routine tasks with software makes our lives more comfortable and efficient. For example, software drives the global economy and transfer of value worldwide making the purchase of goods and the management of finances a seamless experience. Furthermore, at the touch of a few

[1] https://github.com/jpmorganchase/cv4code

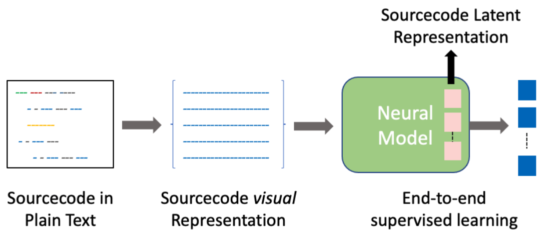

Sourcecode Latent Representation
Neural Model
Sourcecode in Plain Text
Sourcecode *visual* Representation
End-to-end supervised learning

Fig. 1: The proposed CV4Code code understanding pipeline.

buttons on our smartphone we can communicate to friends and family worldwide. Reliable software can also change healthcare outcomes by making it easier for doctors to diagnose disease. A key to accomplishing these feats of automation and being prepared for future complex use-cases is accelerating the software development process without sacrificing software quality, robustness and time-to-market. Augmenting the SDLC with machine learning holds this promise, in which the developer's capabilities are magnified through predictive analytics driven by the vast quantities of exhaust data naturally produced by the SDLC. Machine learning can potentially enhance every stage of the SDLC including requirements gathering, build and test and deployment. For example, AI-driven code auto-completion and enhanced code search are near-term possibilities for enhancing developer productivity with startups and established companies alike productionising such capabilities for mass consumption.

The academic field of MLOnCode explores the application of machine learning techniques for mining the massive amount of sourcecode and associated metadata available in public and private repositories [3]. Indicative tasks in this field include code search using natural language keywords [28] and sourcecode [17] as queries, automated bug finding [21], vulnerability detection [24], design pattern detection [29], program repair [7] and code auto-completion [27]. Core to the field of MLOnCode is the learning of expressive sourcecode latent feature representations ("code vectors") that capture semantics of programs and can be flexibly used in generic machine learning classifiers to support a myriad of downstream tasks, such as code search and repository annotation with semantic keywords. Code is unique from natural language in many respects, for example more distantly spaced tokens may be highly related (*e.g.* opening and closing brackets) and code that looks very similar can have very different behaviour (and vice-versa). Capturing these subtleties and intricacies of sourcecode to learn program semantics requires methods that can understand the underlying context (sequence of tokens) and structure (as presented by syntax parse tree) of the language. Prior methods for sourcecode feature extraction can be differentiated to the extent on which they capture context, structure or both when learning code vectors. For example, early methods treat sourcecode either as a set of independent tokens [2], a sequence of tokens (processed sequentially by an RNN or CNN) or generate an Abstract Syntax Tree (AST) from the code snippet before linearising the tree into vector form, thereby capturing local structure in the snippet [17]. Drawing a parallel between the word2vec model in natural language processing (NLP), Alon *et al.* [5] propose a code2vec alternative that uses the proxy task of method name prediction based on paths in the AST tree to learn expressive code vectors from a shallow neural network architecture. More recent research has explored transformer architectures to learn effective code vectors [31] that capture structure and context. In contrast to prior research, we represent sourcecode in a visual way as a set of *images* that explicitly, through the 2-dimensional spatial representation, present both the code structure and context directly to the learning algorithm. We adapt well-known image processing techniques to process these sourcecode images. We argue that, with no assumption of naturalness in programming language [3], treating sourcecode understanding as a computer vision problem can not only produce more effective code vectors, but can also address key limitations of existing methods such as their inability to featurise partial code snippets and syntactically incorrect code.

In more detail we introduce a series of vision models, including Residual Convolutional Neural Networks (CNNs) [13] and Vision Transformers (ViT) variants [14, 15, 11], adapted for sourcecode understanding (Figure 1). We contribute to the sparse amount of prior research that draws a parallel between successful image understanding models in the field of Computer Vision and their application to representation learning for sourcecode [6, 23, 9]. Different to this closely related prior research, CV4Code does not require any language-specific pre-processing (*e.g.* extraction of syntax parse trees). To represent sourcecode in a visual form, we propose a novel compact encoding of sourcecode as a two dimensional spatial grid of numeric values that represent the characters in the code by their ASCII codepoints. This representation is advantageous over a standard RGB pixel representation of the code for two key reasons: 1) *elimination of redundancy;* for a pixel representation many pixels would be dedicated to encoding a single character; and 2) *fast feature generation:* we find it multiple order of magnitude slower and less scalable to render a pixel representation of code (sub-second) compared to the proposed code representation (sub-millisecond). The computational speed-up enables real-time applications over large codebases, such as code search directly within the Integrated Development Environment (IDE). The proposed image encoding for sourcecode is therefore practical for real-world machine learning pipelines. The CV4Code architecture is designed for learning representations effectively from the ASCII-based codepoint encoded sourcecode images. CV4Code ingests the image representation of code and encodes each pixel as either a one-hot or learnable embedding. The encoded input is subsequently processed by a neural network that learns features expressive for the predictive task *e.g.* the language of code, the task being solved by the code *etc.* And the learned latent embedding from CV4Code can be used as code embeddings for other MLOnCode tasks, similar to VGG features [26] that have been shown to be a powerful and flexible embedding of images for many computer vision tasks.

Our contributions in this paper can be summarised as:

– **CV4Code Deep Neural Models:** We introduce and compare modern deep vision models adapted for language-agnostic sourcecode understanding that learns from a novel ASCII codepoint representation of sourcecode. We report state-of-the-art performance on a public benchmark dataset compared to competitive baselines. Compared to a NLP-inspired Transformer strong baseline, we achieve 4.06% absolute gain in top-1 accuracy on a language-agnostic problem classification task, and 0.011 gain in mAP@R on a similarity based sourcecode retrieval task.
– **ASCII Sourcecode Encoding:** We introduce an ASCII codepoint image representation for sourcecode that efficiently (low redundancy, fast generation) encodes snippets capturing both code structure and context in a single representation.

2 Related work

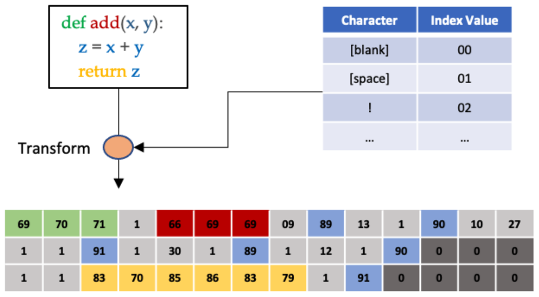
Machine learning for sourcecode analysis aims at learning semantically meaningful representation of the code and then apply the embeddings on downstream tasks, such as code quality detection, code summarisation, defect prediction and code duplication detection [25]. We include the research that are most relevant to our contribution.
AST based representation. Machine learning based intelligent code analysis relies on extracting representative features from sourcecode. The majority of studies leverage structured graphical models for sourcecode, through parse trees. AST carries rich semantic and syntactic information and provides a unique representation of a sourcecode snippet in a given language and grammar [4, 30, 32]. The paper [32] learned jointly from the AST and the sourcecode of programs while relying on language-agnostic features, and performed on code summarisation task on five programming languages. Although the model does not rely on language-specific features, parsing sourcecode to a tree structure is language dependent.
NLP based techniques. A sourcecode sample can be treated as a piece of text. A code snippet can be represented by a vector of frequencies of token occurrences, similar to the bag of word model. The frequently used tokens include regex, keywords and operators [20, 22]. Considering the lack of sequential information retained in the bag of tokens method, a sequence of token method uses the same set of tokens but keeps the order information to form a sequence [22]. Such a token embedding layer is then input into a CNN based model.
Computer vision for code. Despite the fact that more effort has been made on automated sourcecode analysis using machine learning, representing and processing sourcecode in the form of image data is still an under-explored area in the field. Dey *et al.* [9] automatically convert program sourcecode to visual images via an intermediate representation of code created by the LLVM compiler. The ASCII value of the remaining characters are treated as a pixel value in a predefined empty image canvas. Bilgin *et al.* [6] use a coloured image of syntax trees for representing code, where the tokens are plotted in a rectangular shape and

are completed with specific colours to indicate the type and content of the token. The most recent work by Rabin *et al.* [23] on Java code analysis transforms the original code by removing comments and empty lines using a JavaParser tool, and then redacts snapshots of input by replacing any alphanumeric characters in the reformatted input programs with a single letter 'x' to emphasise the structure of code snippets, rather than their content (*e.g.* the specific naming of variables). While the aforementioned research exploited image-based representation for sourcecode, they all require a parser for specific programming language, which cannot process the sentences with incorrect syntax.

3 CV4Code

We propose CV4Code, an end-to-end learning framework for sourcecode understanding by treating sourcecode snippets as *images i.e.* 2-dimensional matrix.

3.1 Sourcecode Representation



```
def add(x, y):
  z = x + y
  return z
```

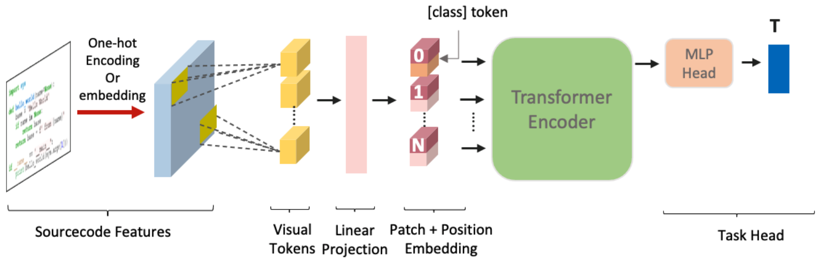
Character	Index Value
[blank]	00
[space]	01
!	02
...	...

69	70	71	1	66	69	69	09	89	13	1	90	10	27
1	1	91	1	30	1	89	1	12	1	90	0	0	0
1	1	83	70	85	86	83	79	1	91	0	0	0	0

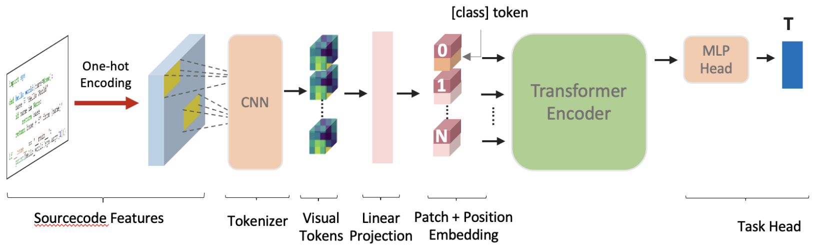
Fig. 2: Example of 2D code representation generation

While the sourcecode of most modern programming languages can be written in plain text from an extensive character set, only a small set of tokens and their composing characters have syntactic and semantic roles. In CV4Code, code snippets are transformed into 2-dimensional (matrix) representation by mapping each printable ASCII character to their unique index values and padding the special *[blank]* token wherever necessary to retain the rectangular shape of the output. The set of valid printable ASCII characters together with the special padding token $\mathbb{V}_c$, $|\mathbb{V}_c| = 96$, consists of the following:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{}|~
[space][blank]
```



(a) CV4Code ViT model overview.



(b) CV4Code Conv-ViT model overview.

Fig. 3: CV4Code transformer model variants.

Figure 2 shows an example of the code representation generation process. Specifically, for a code snippet spanning L lines each with C_l, $l \in 0, ..., L-1$ characters, the transformation is done in three steps:

1. Remove characters not within the valid set, output has $\hat{L}$ lines each with $\hat{C}_l$, $l \in 0, ..., \hat{L}-1$ characters;
2. Map each input character $v_k \in \mathbb{V}_c$ to its index value k;
3. Pad each line to $M = \max_{l=0}^{\hat{L}-1} \hat{C}_l$ long with the index value of *[blank]*, generate the output 2-dimensional code matrix $X \in \mathbb{R}^{L \times M}$.

Compared to human-readable images of sourcecode, *e.g.* screenshots of code snippets, which usually are sparse in semantics and require multiple pixels to represent a single character, the proposed sourcecode representation is compact and do not introduce any unnecessary information other than the blank padding that is required to keep the spatial relations.

As the code image, *i.e.* the compact 2-dimensional sourcecode representation, encodes the character index values which do not form a numerically continuous space, unless otherwise specified, one-hot encoding is used in this work to transform each *pixel* in the code image to a vector of fixed dimension equal to the size of the set of valid characters, i.e. $\mathbf{X} \in \mathbb{R}^{L \times M} \rightarrow \hat{\mathbf{X}} \in \mathbb{R}^{L \times M \times |\mathbb{V}_c|}$.

3.2 Model

We apply and adapt state-of-the-art vision models, including ResNet [13], ViT [14], ViT for small-size datasets (ViT-fsd) [15] and hybrid Convolutional Transformer

(Conv-ViT) [11], on the proposed sourcecode representation. And we show the effectiveness of the proposed method through experiments on a supervised multi-class classification task. Figure 3 shows an overview of the CV4Code transformer model variants. While Figure 3a shows a general architecture of CV4Code-ViT model, differences exist in ViT, ViT-fsd which we briefly describe below.

ViT. We follow [14] and split images into non-overlapping fixed-size patches and prepend a learnable *[class]* embedding whose state at the ViT output serves as the sourcecode representation. This sourcecode representation is then passed to a single-layer MLP head for the classification task.

ViT-fsd. While the same setup as ViT is used, we apply shifted patch tokenization and Locality Self-Attention proposed in [15]. In addition, as the tokenization process creates 4 extra shifted images leading to a largely increased input dimensionality after concatenation, to control the number of parameters in the linear projection layer, instead of one-hot encoding, *i.e.* $\hat{\mathbf{X}} \in \mathbb{R}^{L\times M\times|\mathbb{V}_c|}$, an 32-dimensional learnable embedding is used such that $\hat{\mathbf{X}} \in \mathbb{R}^{L\times M\times 32}$.

Conv-ViT. Shown in Figure 3b. To leverage CNN's inductive bias, *e.g.* locality, similar to [11] and hybrid in [14] we use convolutional layers to create soft visual tokens but keep the use of *[class]* embedding to generate sourcecode representation at the output. Furthermore, as the soft tokenization process does not require fixed-size input, similar to NLP applications of transformers where the input sequence is of variable length, for smaller sourcecode image we append learnable *[pad]* embeddings to the generated visual token sequence, *i.e.* at the output of the CNN, to form equal-length input to the transformer encoder.

3.3 Implementation Details

Variable code snippet size. It is expected that the sourcecode 2-dimensional representation will vary in size. To address this, using the *[blank]* token, we batch up sourcecode snippets of different sizes with *interleaved* padding vertically and constant padding horizontally. While constant padding appends constant values from the end of an array, *interleaved* padding avoids leaving large continuous blank region by inserting *[blank]* tokens between original input code lines. In contrast, if an image exceeds the maximum size limit, we crop and keep the top left corner of the code image, following the raster order to retain most information. For instance, given input of size $L \times M$, $\mathbf{X}_i = [\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{L-1}]$ where $\mathbf{x}_l$ for $l = 0, ..., L-1$ each is a row vector of length M, then the cropped output of size $\hat{L} \times \hat{M}$ is $\hat{\mathbf{X}}_\mathbf{i} = [\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, ..., \hat{\mathbf{x}}_{\hat{L}}]$ where $\hat{\mathbf{x}}_\mathbf{l} = \{x_{l,m}\}_{m=0}^{\hat{M}}$ for $l = 0, ..., \hat{L}-1$. In addition, we pad or limit the input images to be the same size of 96×96 for ResNet, ViT and ViT-fsd, while in Conv-ViT, along with an global minimum of 12×12 and maximum of 96×96 on all batches, we dynamically limit the maximum image size per minibatch at training time, *i.e.* instead of setting a constant maximum limit in all minibatches, it is configured to the 95th-percentile, if smaller than the global maximum, of the width and height independently of the images within the minibatch. Note, while the input code image size in this work is limited to fit for the data distribution, the proposed CV4Code models are not limited by its architecture to process larger snippets.

Dataset	Summary		
	#problems	#samples	#languages
CodeNetBench-Train	237	171300	3
CodeNetBench-Validation	237	21000	3
CodeNetBench-Test	237	21000	3

Table 1: CodeNetBench data summary. Balanced distribution among C++, Python and Java.

Dataset	Summary		
	#problems	#samples	#languages
CodeNetBench-Sim	100	2000	2

Table 2: Similarity evaluation datasets.

Training. We use AdamW [16] optimiser with an learning rate of 10^{-3} and weight decay is set to 0.0001. We also use a 5-epoch warm-up along with a Cosine Learning Rate Annealing. Unless otherwise specified, all models are trained for 100 epochs and the model with the highest validation accuracy is selected to report results on the test set. Our model is implemented in PyTorch and trained on 1x NVIDIA V100 GPU with a batch size of 256.

4 Experimental Evaluation

In order to benchmark the performance and capabilities of our proposed framework, we conduct experiments on a real-world sourcecode dataset and compare against a set of competitive baselines, including character-, token- and AST-based representations respectively. Furthermore, test results are reported on Code Classification and Code Similarity tasks. The goal of these experiments is to answer the following research questions :

- RQ1: How well does CV4Code perform on the tasks in comparison to baselines using alternative forms of source code representation?
- RQ2: How scalable and flexible is CV4Code to baselines using alternative forms of source code representation?
- RQ3: How useful are the latent features learnt by CV4Code for alternate downstream tasks?

4.1 Setup

Dataset. We use CodeNet [22], a high-quality real-world dataset with code samples scraped from online coding platforms. The dataset provides code samples submitted by students and developers from across the world, resulting in a sundry pool of source code. The dataset is categorised based on the problems presented in the platform and for each such problem it provides the submissions

in multiple programming languages. Our choice of CodeNet stems from the fact that we aim to benchmark the sourcecode understanding capability of CV4Code against baseline models in both language-agnostic and language-specific setups, and high-quality CodeNet benchmark set supporting 3 popular languages, including C++, Java and Python, makes it the ideal choice. We use the curated benchmark set of CodeNet [22] as it is made to be challenging with duplicated and dead code samples filtered. First, we extracted a multilingual set composed of code solutions to 237 overlapping *problem_id*s from C++1400, Python800 and Java250 and it is split into train, validation and test sets following 80%, 10% and 10% sample distributions for each *problem_id*. For convenience, we name them CodeNetBench-Train/Validation/Test. Furthermore, for the code similarity retrieval task, we test on *CodeNetBench-Sim* set, in which we randomly sample 100 problems and each problem with 10 code snippets in C++ and Python respectively, *i.e.* 20 code snippets per problem, from CodeNetBench-Test. Finally, we create One-versus-All test pairs, *i.e.* each test sample is paired with all other samples and positive pairs are those of the same *problem_id*.
Tasks. To evaluate the efficacy of our proposed framework, we evaluate on two tasks as follows:

- Code Classification: the goal is to classify source code samples based on their respective programming problem i.e. *problem_id*. Code samples belonging to the same programming problem would have high structural and semantic information overlap. As a result, it provides a solid ground for comparing the effectiveness of different source code representations.
- Code Similarity: the goal is to compare the efficacy of the various latent sourcecode representations for retrieving *similar* code samples.

Table 1 and 2 summarise the datasets classification model training, evaluation and similarity task evaluation.
Loss function. As for the loss function for *problem_id*s classification, we adopt Additive Angular Margin (AAM) Softmax loss [8], as shown in Equation 1, which has been shown to perform well by explicitly optimising similarity for intra-class samples and diversity for inter-class samples.

$$L = -\frac{1}{N}\sum_{i=1}^{N} \ln \frac{\exp\{s\cdot\cos(\theta_{y_i,i}+m)\}}{\exp\{s\cdot\cos(\theta_{y_i,i}+m)\}+\sum_{j\neq y_i}\exp\{s\cdot\cos(\theta_{j,i})\}}) \tag{1}$$

where $\theta_{y_i,j}$ is the angle between i-th sample to the y_i-th class and N is the batch size. We use angular margin penalty $m = 0.2$ and feature scale $s = 30$. For consistency and fairness, the same loss function is used in all models trained on *problem_id* classification.

4.2 Baseline Methods

In this section we describe baseline methods to which we compare the proposed method against, including language agnostic and language specific ones. Specifically, we categorise a method as language-agnostic or language-specific 1) if any

Input Size	fc0	fc1	fc2	Output
N	128	256	512	237

Table 3: Bag of Character (BoC) and SPTR-Java MLP model configurations. ReLU activation and BatchNorm are used in fc layers. N = 95, 256 and 512 respectively in BoC, SPTR-Java-S and SPTR-Java-L.

Model	Vocab	Depth	Hidden Size D	MLP size	Heads	Params
a-Transformer	30K	12	128	512	4	7.1M
k-Transformer	120	12	128	512	4	3.3M

Table 4: Token transformer model configurations. Learnable position embedding is used.

language-specific pre- or post-processing, including feature extraction, technique is required; 2) if any language-specific assumption is imposed on the model.

Language Agnostic Models

Bag of Characters. (BoC) A code sample is represented by the relative frequencies of character occurrence. Specifically, the feature vector of a code snippet is formed by counting the number of occurrences of each valid character introduced in 3.1 (excluding *[blank]*). Table 3 summarises the configuration of the MLP network that was selected after experiments with an array of setups. The training strategy described in Section 3.3 is used.

Token Transformer. Similar to CodeBERT [10], relying on the assumption of naturalness in programming language, we build state-of-the-art NLP Transformer with text-based tokens input as baseline. Specifically, we learn two Token Transformer models via supervised training on the *problem_id* task, one with input of all tokens, including *all* operators and words and another with only 120 combined *keywords* and key operators from Python, Java and C++ provided in [22], which we call a-Transformer and k-Transformer respectively. To avoid noisy and sparse input embedding for a-Transformer, we extract a vocabulary in which each token occurs at least twice in the training dataset. In addition, both models have maximum input token length of 512. Table 6 summarises the configurations of the Transformers that was empirically selected. While the same training strategy described in 3.3 is followed, due to the higher model memory footprint compared to other models, we use a smaller batch size of 32.

Language Specific Models

Simplified Parse Tree Relation. (SPTR) Leveraging Simplified Parse Tree (SPT) features originally presented in [17], SPTR directly exploits the underlying structure from code snippets. We extract SPT and build a vocabulary from

conv0	conv1	conv2	conv3	fc	Output
$16, 2$	$\begin{bmatrix}64\\64\end{bmatrix} \times 2,\ 2$	$\begin{bmatrix}128\\128\end{bmatrix} \times 2,\ 2$	$\begin{bmatrix}256\\256\end{bmatrix} \times 2, 1$	128	237

Table 5: CV4Code-ResNet model configuration. Conv layer weights are annotated as number of filters and stride step size. 7×7 kernel is used in conv0 and 3×3 in others. 3×3 with stride 2 and 6×6 (global) max_pool is used after conv0 and conv3. Total #params=3.25M.

Model	Patch size	Params
CV4Code-ViT-S	16×16	5.32M
CV4Code-ViT-L	8×8	2.98M
CV4Code-ViT-fsd-S	16×16	13.97M
CV4Code-ViT-fsd-L	8×8	4.58M

Table 6: CV4Code-ViT and CV4Code-ViT-fsd configurations. All configurations use depth of 8, hidden size of 128, MLP size of 512 with 4 heads. Learnable position embedding is used.

Model	Convolutional Tokenizer	Visual Tokens	Params
Conv-ViT-S	$\begin{bmatrix}7 \times 7, 64\end{bmatrix} \times 2$, stride 2	49	2.35M
Conv-ViT-L	$\begin{bmatrix}3 \times 3, 64\end{bmatrix} \times 3$, stride 1	169	1.83M

Table 7: Conv-ViT configurations. All configurations use depth of 8, hidden size of 128, MLP size of 512 with 4 heads. Each convolutional layer is followed by a 2×2 max_pool layer with stride of 2. Fixed Sinusoidal position embedding is used.

all training Java code samples. Then a sparse binary count vector is extracted from each SPT that defines the existence of a particular vocabulary from a SPT. Finally dense feature vectors are obtained through Truncated Single Value Decomposition (tSVD). We experiment with the two MLP models, as summarised in Table 3, with tSVD output dimensions set to 256 and 512 respectively (explained variance ratios of 0.768 and 0.834). We train and report results on Java samples in CodeNetBench-Train and CodeNetBench-Test, following the same training strategy described in section 3.3.

4.3 CV4Code setup

Table 5 reports the model configuration for CV4Code-ResNet, we experiment with various configurations and report result from the model with the best validation accuracy. For ViT and ViT-fsd, we experiment with two different patch sizes, *i.e.* 16×16 and 8×8, which result in 36 and 144 visual tokens respectively, given input sourcecode image of size 96×96. For Conv-ViT, we compare two convolutional soft tokenization setups which result in similar length of visual tokens. Table 7 summarises the configurations for all vision transformer variants.

Model	Multilingual		Java-only	
	Top-1	Top-5	Top-1	Top-5
BoC	80.97	90.16	80.56	89.74
k-Transformer	90.30	95.42	89.54	94.97
a-Transformer	93.58	96.63	94.04	97.40
SPTR-Java-S	-	-	91.09	96.98
SPTR-Java-L	-	-	92.95	96.78
ResNet	92.93	96.50	91.17	95.50
ViT-S	85.45	93.64	80.50	90.95
ViT-L	92.85	96.86	90.27	95.46
ViT-fsd-S	86.04	93.80	80.60	90.80
ViT-fsd-L	92.27	96.47	88.99	94.49
Conv-ViT-S	96.08	98.45	94.63	98.01
Conv-ViT-L	**97.64**	**98.99**	**97.13**	**98.79**

Table 8: *problem_id* classification results on CodeNetBench-Test.

Model	mAP@R
a-Transformer	0.980
ResNet	0.983
Conv-ViT-L	**0.991**

Table 9: Code similarity evaluation result on CodeNetBench-Sim.

4.4 Results

Evaluation Metrics. For classification tasks, top-1 and top-5 accuracy are reported as the evaluation metrics on CodeNetBench-Test. For code similarity, we consider a retrieval task where a code snippet is used as query to search for similar snippets and mAP@R[19] is reported as the main evaluation metric.

Language-agnostic Classification. As summarised in Table 8, Conv-ViT-L outperforms all other models in multilingual test, including the strong NLP-based a-Transformer. We attribute the gain of CV4Code over k-Transformer to exploitation of the contextual and structural information in the spatial relationships, which is not directly available through the sequential token input in k-Transformer. Comparing CV4Code ResNet and transformer variants, ViT and ViT-fsd variants all perform worse than ResNet. This potentially implies that the inductive bias, *e.g.* locality, associated with convolutional networks are critical for sourcecode understanding from our proposed sourcecode image representation. This is further supported by the gain obtained in Conv-ViT-S/L which inherit the inductive bias through its soft convolutional tokenizer and leverage the expressiveness of the Transformer network. In addition, it is noticed that the size of patch and the subsequently generated visual tokens have a strong influence on the performance of all CV4Code transformer variants, which is exhibited

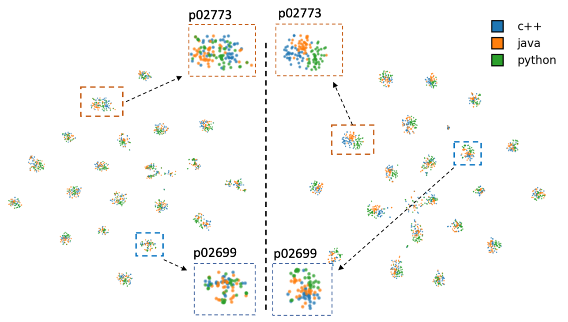
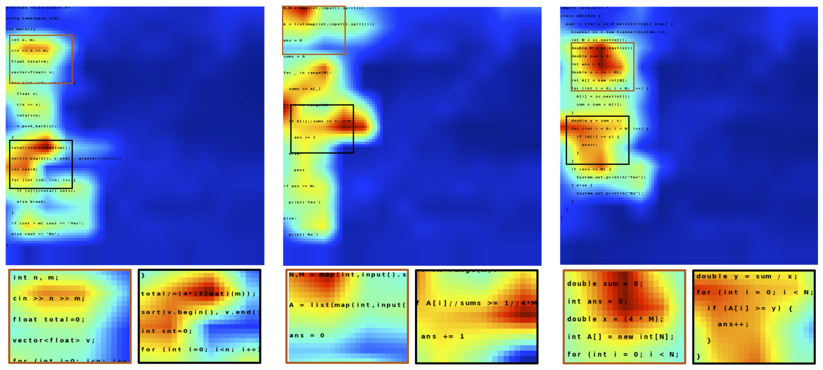


Fig. 4: t-SNE 2D projection of **left**: Conv-ViT-L and **right**: a-Transformer embeddings. Colour-labeled by unique programming *language*s.



```
int n, m;
cin >> n >> m;
float total=0;
vector<float> v;
```

```
}
total/=(4*(float)(m));
sort(v.begin(), v.end(
int cnt=0;
for (int i=0; i<n; i++
```

```
N,M = map(int,input().s
A = list(map(int,input(
ans = 0
```

```
f A[i]//sums >= 1//4*M
ans += 1
```

```
double sum = 0;
int ans = 0;
double x = (4 * M);
int A[] = new int[N];
for (int i = 0; i < N;
```

```
double y = sum / x;
for (int i = 0; i < N;
  if (A[i] >= y) {
    ans++;
  }
}
```

Fig. 5: Attention maps (rollout) of Conv-ViT-L. 1) C++, 2) Python, 3) Java

through the gap between all -S/-L model pairs. In addition, for Conv-ViT-L, we also experimented with 2- and 8-head attentions achieving 97.65% and 96.98% top-1 accuracy scores respectively, showing minor variations.

Language-specific (Java) Classification The Java-only column in Table 8 summaries model test results on Java samples in CodeNetBench-Test. Although SPTR-Java-L/S both achieve strong performance, which demonstrates the effectiveness of the input features that exploit the underlying code structures via SPTs, it is outperformed by a-Transformer by 1.09% in terms of Top-1 accuracy. And Conv-ViT-L, as a language-agnostic model trained on C++, Python and Java, still achieves the strongest result overall with a 97.13% Top-1 accuracy.

Code Similarity We test the learned embeddings from the models that achieve strong results on the classification task, with respect to code similarity and report similarity mAP@R scores on datasets summarised in Table 2. We extract *[class]* embedding at the transformer output from a-Transformer, pre-ReLU bottleneck output from ResNet and sequence pooled embedding from transformer output from Conv-ViT-L. Cosine similarity is used to compute the pairwise similarity of the embeddings. Test results are shown in Table 9. We observe that CV4Code models, including ResNet and Conv-ViT-L, outperform a-Transformer on this task, implying that the learned embeddings from proposed models are highly discriminative and encodes the *semantics* of sourcecode.

4.5 Ablation studies

Influence of Programming Languages. We look at the influence of programming languages on the latent representations from Conv-ViT-L, compared with a-Transformer. t-SNE[18] is used for projection. As shown in figure 4, while both embedding spaces show clusters formed with respect to *problem_id*s, we observe slightly more obvious separation with respect to *languages* in a-Transformer. Given that distinctive syntax, use of operators, coding styles and naming conventions are often used in each programming language, this implies that models with text-based token input sequence is potentially more sensitive to the underlying programming language than the proposed approach.
Attention maps. Following [1], we compute the average attention weights across all heads and recursively multiply them over all layers. We notice that Conv-ViT-L globally attends to regions that are semantically important. For example, in Figure 5, with three different code snippets solving the same task, despite the syntactic and language differences, the model attends to code sections that are functionally near identical and highly relevant to the task. **Future Studies**. Following recent advancement of self-supervised learning for Vision models [12], we would like to scale up CV4Code with the abundance of unlabelled sourcecode snippets in the public domain. With such a setup, we look to establish a study of self-supervised approaches with the proposed CV4Code method and compare to NLP approaches,*e.g.* BERT, for additional MLOnCode problems. Furthermore, to capture richer semantics in the latent space and test on more generic downstream applications, *e.g.* code quality assessment, we plan to extend the training framework to include multi-task learning configurations.

5 Conclusion

In this work, we propose an idea for sourcecode understanding via a novel visual representation. Compared to traditional syntax parse tree or token based methods, the proposed approach is programming language agnostic and does not depend on syntax correctness in the preprocessing stage. Finally, along with a thorough study of vision models applied on the proposed representations and a comparison with syntax tree and NLP-based approaches, using a Compact Convolution Transformer CV4Code model, we show state-of-the-art performance in terms of both classification and code similarity retrieval.

References

1. Abnar, S., Zuidema, W.: Quantifying attention flow in transformers. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4190–4197. Association for Computational Linguistics, Online (Jul 2020). https://doi.org/10.18653/v1/2020.acl-main.385, https://aclanthology.org/2020.acl-main.385
2. Allamanis, M., Sutton, C.: Mining source code repositories at massive scale using language modeling. In: 2013 10th Working Conference on Mining Software Repositories (MSR). pp. 207–216 (2013). https://doi.org/10.1109/MSR.2013.6624029
3. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**(4), 81 (2018)
4. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=H1gKYo09tX
5. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290353, https://doi.org/10.1145/3290353
6. Bilgin, Z.: Code2image: Intelligent code analysis by computer vision techniques and application to vulnerability prediction. arXiv preprint arXiv:2105.03131 (2021)
7. Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.N., Poshyvanyk, D., Monperrus, M.: Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering **47**(9), 1943–1959 (2021). https://doi.org/10.1109/TSE.2019.2940179
8. Deng, J., Guo, J., Xue, N., Zafeiriou, S.: Arcface: Additive angular margin loss for deep face recognition. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 4690–4699 (2019)
9. Dey, S., Singh, A.K., Prasad, D.K., Mcdonald-Maier, K.D.: Socodecnn: Program source code for visual cnn classification using computer vision methodology. IEEE Access **7**, 157158–157172 (2019)
10. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics, Online (Nov 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139, https://aclanthology.org/2020.findings-emnlp.139
11. Hassani, A., Walton, S., Shah, N., Abuduweili, A., Li, J., Shi, H.: Escaping the big data paradigm with compact transformers. ArXiv **abs/2104.05704** (2021)
12. He, K., Chen, X., Xie, S., Li, Y., Dollár, P., Girshick, R.: Masked autoencoders are scalable vision learners. arXiv preprint arXiv:2111.06377 (2021)
13. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
14. Kolesnikov, A., Dosovitskiy, A., Weissenborn, D., Heigold, G., Uszkoreit, J., Beyer, L., Minderer, M., Dehghani, M., Houlsby, N., Gelly, S., Unterthiner, T., Zhai, X.: An image is worth 16x16 words: Transformers for image recognition at scale (2021)
15. Lee, S.H., Lee, S., Song, B.C.: Vision transformer for small-size datasets. arXiv preprint abs/2112.13492 (2021)
16. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. In: ICLR (2019)

17. Luan, S., Yang, D., Barnaby, C., Sen, K., Chandra, S.: Aroma: Code recommendation via structural code search. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019)
18. van der Maaten, L., Hinton, G.: Visualizing data using t-sne. Journal of Machine Learning Research **9**(86), 2579–2605 (2008), http://jmlr.org/papers/v9/vandermaaten08a.html
19. Musgrave, K., Belongie, S., Lim, S.N.: A metric learning reality check. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J.M. (eds.) Computer Vision – ECCV 2020. pp. 681–699. Springer International Publishing, Cham (2020)
20. Ochodek, M., Hebig, R., Meding, W., Frost, G., Staron, M.: Recognizing lines of code violating company-specific coding guidelines using machine learning. Empirical Software Engineering **25**(1), 220–265 (2020)
21. Pradel, M., Sen, K.: Deepbugs: A learning approach to name-based bug detection. Proc. ACM Program. Lang. **2**(OOPSLA) (Oct 2018). https://doi.org/10.1145/3276517, https://doi.org/10.1145/3276517
22. Puri, R., Kung, D., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Finkler, U.: Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks (2021)
23. Rabin, M.R.I., Alipour, M.A.: Encoding program as image: Evaluating visual representation of source code. arXiv preprint arXiv:2111.01097 (2021)
24. Russell, R.L., Kim, L.Y., Hamilton, L.H., Lazovich, T., Harer, J.A., Ozdemir, O., Ellingwood, P.M., McConley, M.W.: Automated vulnerability detection in source code using deep representation learning. CoRR **abs/1807.04320** (2018), http://arxiv.org/abs/1807.04320
25. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Sarro, F.: A survey on machine learning techniques for source code analysis. arXiv preprint arXiv:2110.09610 (2021)
26. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: International Conference on Learning Representations (2015)
27. Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J.V., Allamanis, M.: Fast and memory-efficient neural code completion. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021. pp. 329–340. IEEE (2021). https://doi.org/10.1109/MSR52588.2021.00045, https://doi.org/10.1109/MSR52588.2021.00045
28. Yan, S., Yu, H., Chen, Y., Shen, B., Jiang, L.: Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 344–354 (2020). https://doi.org/10.1109/SANER48275.2020.9054840
29. Zanoni, M., Arcelli Fontana, F., Stella, F.: On applying machine learning techniques for design pattern detection. J. Syst. Softw. **103**(C), 102–117 (May 2015). https://doi.org/10.1016/j.jss.2015.01.037, https://doi.org/10.1016/j.jss.2015.01.037
30. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 783–794. IEEE (2019)
31. Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., Günnemann, S.: Language-agnostic representation learning of source code from structure and context. In: International Conference on Learning Representations (ICLR) (2021)

32. Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., Günnemann, S.: Language-agnostic representation learning of source code from structure and context. arXiv preprint arXiv:2103.11318 (2021)