Learning Lexical Features of Programming Languages from Imagery Using Convolutional Neural Networks

Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, Erik Linstead
Machine Learning and Assistive Technology Lab
Schmid College of Science and Technology
Chapman University
Orange, California
{ott109,atchi102,harna100,best120,ander427,firma103}@mail.chapman.edu,linstead@chapman.edu

ABSTRACT

We demonstrate the ability of deep architectures, specifically convolutional neural networks, to learn and differentiate the lexical features of different programming languages presented in coding video tutorials found on the Internet. We analyze over 17,000 video frames containing examples of Java, Python, and other textual and non-textual objects. Our results indicate that not only can computer vision models based on deep architectures be taught to differentiate among programming languages with over 98% accuracy, but can learn language-specific lexical features in the process. This provides a powerful mechanism for carrying out program comprehension research on repositories where source code is represented with imagery rather than text, while simultaneously avoiding the computational overhead of optical character recognition.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Software and its engineering → Software libraries and repositories**;

KEYWORDS

deep learning, convolutional neural networks, program syntax

ACM Reference Format:
Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, Erik Linstead. 2018. Learning Lexical Features of Programming Languages from Imagery Using Convolutional Neural Networks. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196359

1 INTRODUCTION

In recent years, the state-of-the-art in computer vision techniques have converged on deep learning solutions based on artificial neural networks. In particular, convolutional neural networks (CNNs) have demonstrated the ability to learn high-order features in imagery which can be leveraged for both supervised and unsupervised machine learning tasks. While this has led to a wide variety of applications in areas such as medical informatics and autonomy, the inherently textual nature of source code has left CNNs relatively unexplored in the software engineering community.

Despite the textual nature of source code, substantial volumes of software data remain embedded in images and video as part of technical tutorials available on the Internet. This data provides an opportunity for new directions in program comprehension research, particularly the ability of computer vision models to learn visual differences, such as syntax, among programming languages in the same way that human eyes perceive these features when looking at code. If computer vision models are capable of learning such features, these models can form the basis of new techniques for the automated understanding of code in images and videos that is not predicated on optical character recognition to first convert the data into text.

In this paper we apply, for the first time, CNNs to the task of learning lexical features from image-based representations of programs, focusing on Java and Python for a pilot study. Leveraging a training set of 17,500 hand-labeled images, we are able to achieve 98.73% cross-validated accuracy distinguishing Java and Python, and 92.50% accuracy distinguishing Java and Python from non-code data. Using class activation mapping (CAM) [16] as a visualization technique, we can see that our CNN models are learning low-level lexical features of programming languages as part of the training process, similar to cues a human uses to differentiate one language from another.

2 DATA

As a first step in exploring the suitability of CNNs for identifying lexical features of programming languages, we cultivated a corpus of 100 tutorials consisting of approximately 50 hours of video from YouTube. A diverse set of Integrated Development Environments (IDEs), text editors, font sizes, and text colors appear in the dataset. We focused on the Java and Python programming languages for the experiments described here. All 100 videos were downloaded to our server and segmented into a discrete image set by sampling at a rate of one frame per second. This resulted in 160,500 unlabeled images. These images were then filtered manually to exclude frames with code examples that were obstructed or obscured in any way, or contained handwritten code (such as on a whiteboard). This

ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196359

Binary Classification	Training Set			Testing Set		
	Java	Python		Java	Python	
Java vs Python	4,853	2,514		1,215	630	
Categorical Classification	Training Set			Testing Set		
	Java	Python	NC	Java	Python	NC
Java vs Python vs NC	4,853	2,514	5,495	1,215	630	1,370

Table 1: Average data set size of each cross validation fold.

step was taken to ensure that the CNNs could leverage full textual context as part of the learning process. In the future it would be worthwhile to explore the ability of the models to generalize to obscured or handwritten code.

As with all supervised learning methods, the ability of CNNs to achieve high classification accuracy is predicated on the availability of accurately labeled training data. To label as many images as accurately and efficiently as possible, help was solicited from approximately 50 students enrolled in freshman and sophomore-level courses focusing on programming in Java and Python. Students were asked to label images as coming from one of three categories: containing Java code, containing Python code, or not containing either Java or Python. This process resulted in a final image set of 17,500 frames. Due to differing resolutions in our video corpus, as well as a need for uniform input sizes in the first layer of our neural networks, all images were rescaled to $500x500$ pixels.

3 METHODS

Structured input data, such as images, lose their spatial relationships when passed through traditional fully-connected, feed-forward artificial neural networks (ANNs). This is problematic for applications such as computer vision since image features are comprised of groups of pixels. Convolutional Neural Networks represent an alternative ANN architecture that is able to maintain spatial relations between pixels by convolving the input space with a multidimensional weight matrix, commonly referred to as a filter. Training CNNs with backpropagation was first proposed by LeCunn et al. [7]. CNNs use a shared weight paradigm to reduce the number of trained parameters, and as a result scale better compared to their fully-connected counterparts. Weight sharing also builds translational invariance into the learned model, so that features can be recognized despite their specific location in the image.

In this paper, we leverage the VGG [11] network, a popular CNN architecture, to distinguish between Java and Python in video frames. The VGG network has a convenient architecture in which multiple convolutional operations occur in succession, followed by a max pooling layer, which has the effect of downsampling high-dimensional pixel spaces. After these layers have been repeated several times, a fully connected output layer, typically implementing a softmax function, is added. For our experiments, the VGG architecture was implemented in Python using the Keras API with a TensorFlow backend using two NVidia P100 GPUs with 16 GBs of memory and 3,584 CUDA cores each.

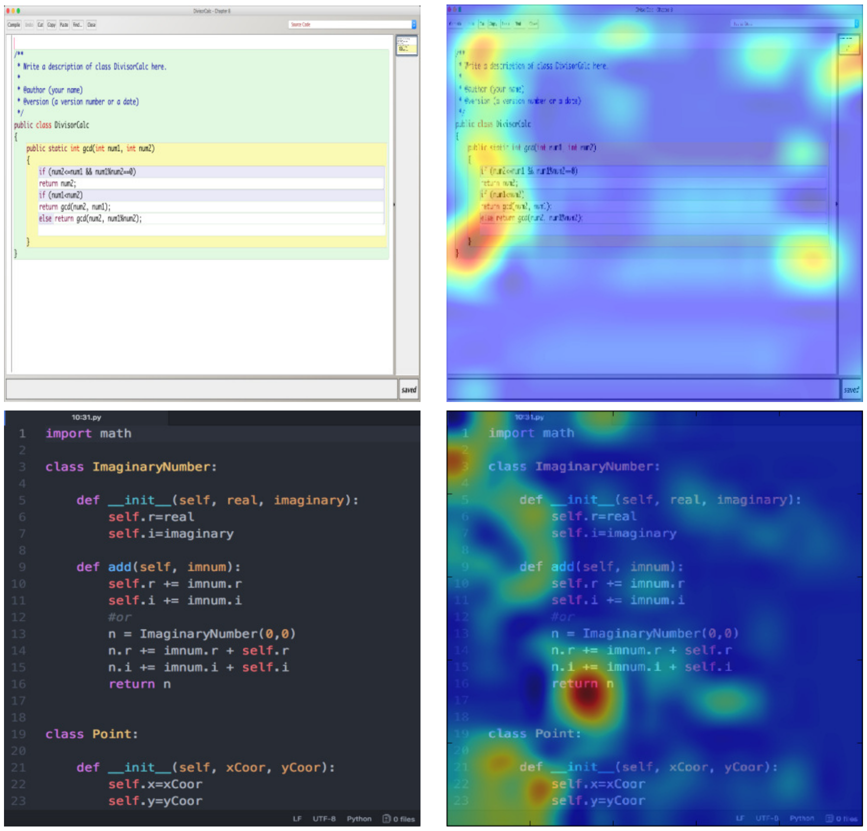
4 RESULTS

Convolutional neural networks were trained using 5-fold cross-validation for the two experiments detailed in Table 1. Each convolutional model took, on average, 2.5 hours per fold to train, for a total of 37.5 computing hours of training for all folds in all models. In practice, the overall time was decreased by training models in parallel by taking advantage of multiple GPUs on our deep learning server.

The mean accuracies of the 5-fold cross-validation experiments for each classifier are detailed below. A mean accuracy of 98.73% (median 98.75%) is achieved on the binary classification task of predicting Java versus Python typeset visible code. Including a third category of no code, yields an accuracy of 92.50% (median 92.60%). This accuracy was achieved when both Java and Python datasets contained only visible typeset code.

Figure 1 shows CAM results on correctly predicted Java code image frames. Additional CAM figures are available in the supplementary material. The heatmap produced by CAM can be interpreted by the degree of redness in a given region. The more red a region is, the more weight the network associates with features in that area to formulate its output prediction. In Figure 1, the left column shows examples from the test set, while the right column shows the CAM overlaid on the corresponding test image. The first row shows an example of correctly predicted Java typeset code, while the second shows a Python image frame. Visual analysis of the CAM result images reveals the network's preference towards Java and Python specific features such as method and class declarations, semicolons, and curly brackets. For example, the Java example in Figure 1 shows the network's strong preference for curly brackets when predicting Java as the category. The Python image directly below it shows strong preference for the def keyword as well as Python's indentation pattern. The results of Figure 1 show the network is capable of learning lexical and contextual features of image code frames.

To ensure our method can distinguish between Java and Python in homogeneous settings, we present the CAM results in Figure 2. Like Figure 1, the left column shows examples from the test set, while the right column shows the CAM overlaid on the corresponding test image. Both Java and Python code snippets are of the Quicksort algorithm. Each line is semantically the same, only differing in the languages appropriate syntax. The first row shows an example of correctly predicted Java typeset code, while the second shows a Python image frame. The Java CAM image shows the network identifies curly brackets, semicolons, backslashes for comments, and keywords such as public and private. The Python CAM image highlights then end of lines, indicating an absence of semicolons. These results show that convolutional networks, such



```
/**
 * Write a description of class DivisorCalc here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class DivisorCalc
{
    public static int gcd(int num1, int num2)
    {
        if (num2<=num1 && num1%num2==0)
        return num2;
        if (num1<num2)
        return gcd(num2, num1);
        else return gcd(num2, num1%num2);
    }
}
```

```
import math

class ImaginaryNumber:

    def __init__(self, real, imaginary):
        self.r=real
        self.i=imaginary

    def add(self, imnum):
        self.r += imnum.r
        self.i += imnum.i
        #or
        n = ImaginaryNumber(0,0)
        n.r += imnum.r + self.r
        n.i += imnum.i + self.i
        return n


class Point:

    def __init__(self, xCoor, yCoor):
        self.x=xCoor
        self.y=yCoor
```

Figure 1: CAM results on correctly predicted Java (top) and Python (bottom) code image frames. The left column shows the normal test image. The right column shows the CAM results overlaid on the test image.

as the ones used in this study, are capable of learning, visually, lexical differences between programming languages.

Using CAM, we are able to visualize what regions of input images the network attends to when making its classification prediction. This allows us to ensure the network is learning features directly related to the language's syntax and not other circumstantial features contained in the images (IDE/text editor features). Additional CAM results, as well as the convolutional architecture used for the artificial neural networks in this study, are available in the supplementary material: https://github.com/mlat/icpc.

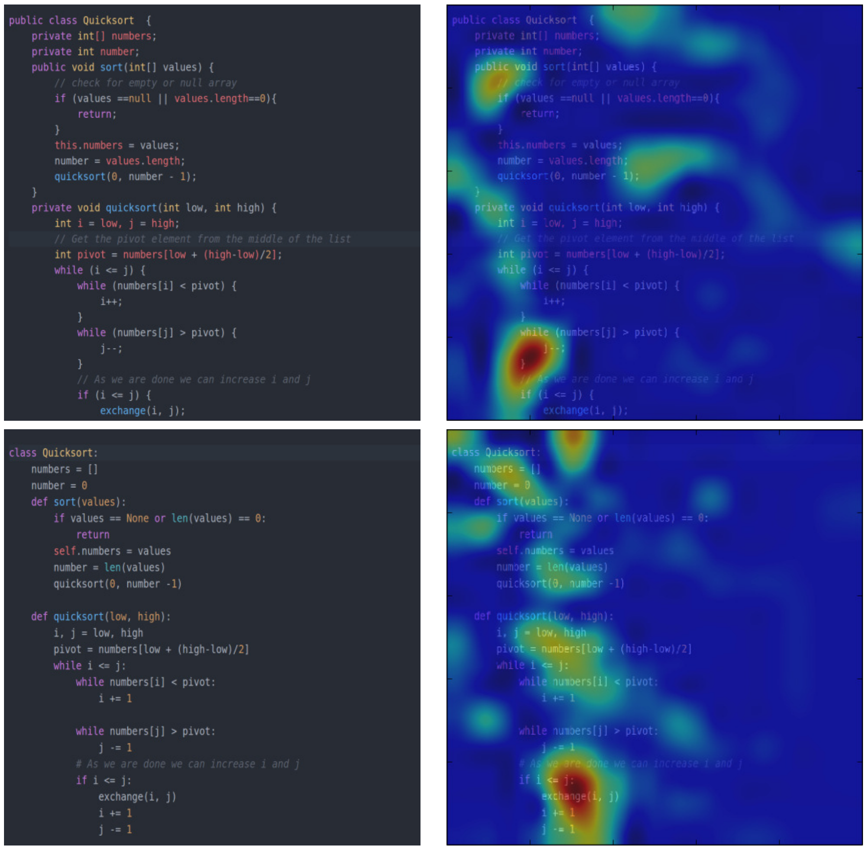
5 RELATED WORKS

The application of deep learning to the study of natural language represented as text was conducted in [2]. The authors leveraged a single CNN architecture to predict part-of-speech tags, chunks, named entity tags, as well as other semantic attributes given an input sentence. The authors in [6] use recurrent neural network grammars to identify and learn heads of phrases in order to determine the syntactic category of a natural language phrase. In [1], the authors propose a modular ANN architecture for lexical analysis of natural language in the form of a continuous input stream.

Digital image processing in the code recognition domain is outlined in [10]. In this study, the authors look to identify Java code in video frames through the application of OCR to candidate sub frames. In [14], the authors use language specific statistical modeling to identify code regions appearing across frames. In [9], deep-learning is applied to this domain through the application of convolutional neural networks to Java programming tutorials. This approach allows for a more scalable solution to video indexing while still maintaining accuracy. In this study, we extend this deep learning approach, focusing on building a single model that can differentiate between multiple languages while learning lexical features in the process.

The in-depth analysis of source code samples, although not in the form of images, has been explored in depth over the past decade. The study of vocabulary trends throughout Java software development is conducted in [8]. Others have sought to classify languages given source code samples. In [15], the authors present a maximum entropy classifier. The work in [12] employs a support vector machine based classifier while a statistical analysis of program features is conducted in [5]. In [4], a Multinomial Naive Bayes classifier is used and a modified Kneser-Ney discounting classifier is presented in [13]. The application of a CNN to language classification comes in [3]. This paper demonstrates the efficiency and accuracy of applying CNNs to the problem of classifying textual source code.

While previous work shares our goal of leveraging language features to classify programming languages, we believe our study



```java
public class Quicksort  {
    private int[] numbers;
    private int number;
    public void sort(int[] values) {
        // check for empty or null array
        if (values ==null || values.length==0){
            return;
        }
        this.numbers = values;
        number = values.length;
        quicksort(0, number - 1);
    }
    private void quicksort(int low, int high) {
        int i = low, j = high;
        // Get the pivot element from the middle of the list
        int pivot = numbers[low + (high-low)/2];
        while (i <= j) {
            while (numbers[i] < pivot) {
                i++;
            }
            while (numbers[j] > pivot) {
                j--;
            }
            // As we are done we can increase i and j
            if (i <= j) {
                exchange(i, j);
```

```python
class Quicksort:
    numbers = []
    number = 0
    def sort(values):
        if values == None or len(values) == 0:
            return
        self.numbers = values
        number = len(values)
        quicksort(0, number -1)

    def quicksort(low, high):
        i, j = low, high
        pivot = numbers[low + (high-low)/2]
        while i <= j:
            while numbers[i] < pivot:
                i += 1

            while numbers[j] > pivot:
                j -= 1
            # As we are done we can increase i and j
            if i <= j:
                exchange(i, j)
                i += 1
                j -= 1
```

Figure 2: CAM results on correctly predicted Java (top) and Python (bottom) code image frames, of the Quicksort algorithm. The left column shows the normal test image. The right column shows the CAM results overlaid on the test image.

represents the first use of deep learning to do this using native images instead of text.

REFERENCES

[1] Chun-Hsien Chen and V. Honavar. 1999. A neural-network architecture for syntax analysis. *IEEE Transactions on Neural Networks* 10, 1 (1999), 94–114. https://doi.org/10.1109/72.737497

[2] Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08).* ACM, New York, NY, USA, 160–167. https://doi.org/10.1145/1390156.1390177

[3] S. Gilda. 2017. Source code classification using Neural Networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE).* 1–6. https://doi.org/10.1109/JCSSE.2017.8025917

[4] Jyotiska Nath Khasnabish, Mitali Sodhi, Jayati Deshmukh, and G. Srinivasaraghavan. 2014. Detecting Programming Language from Source Code Using Bayesian Learning Techniques. In *Machine Learning and Data Mining in Pattern Recognition*, Petra Perner (Ed.). Springer International Publishing, Cham, 513–522.

[5] David Klein, Kyle Murray, and Simon Weber. 2011. Algorithmic Programming Language Identification. *CoRR* abs/1106.4064 (2011). arXiv:1106.4064 http://arxiv.org/abs/1106.4064

[6] Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, Graham Neubig, and Noah A. Smith. 2016. What Do Recurrent Neural Network Grammars Learn About Syntax? *CoRR* abs/1611.05774 (2016). arXiv:1611.05774 http://arxiv.org/abs/1611.05774

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[8] Erik Linstead, Lindsey Hughes, Cristina Lopes, and Pierre Baldi. 2009. Exploring Java software vocabulary: A search and mining perspective. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation.* IEEE Computer Society, 29–32.

[9] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A Deep Learning Approach to Identifying Source Code in Images and Video. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR).*

[10] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch!: extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 261–272.

[11] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[12] Secil Ugurel, Robert Krovetz, and C. Lee Giles. 2002. What's the Code?: Automatic Classification of Source Code Archives. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02).* ACM, New York, NY, USA, 632–638.

[13] J. K. v. Dam and V. Zaytsev. 2016. Software Language Identification with Natural Language Classifiers. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 624–628. https://doi.org/10.1109/SANER.2016.92

[14] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM, 98–111.

[15] Shaul Zevin and Catherine Holzem. 2017. Machine Learning Based Source Code Classification Using Syntax Oriented Features. *CoRR* abs/1703.07638 (2017). arXiv:1703.07638 http://arxiv.org/abs/1703.07638

[16] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning deep features for discriminative localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2921–2929.