# A Deep Learning Approach to Identifying Source Code in Images and Video

Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, Erik Linstead
Machine Learning and Assistive Technology Lab
Schmid College of Science and Technology
Chapman University
Orange, California
{ott109,atchi102,harna100}@mail.chapman.edu,{abergh,linstead}@chapman.edu

## ABSTRACT

While substantial progress has been made in mining code on an Internet scale, efforts to date have been overwhelmingly focused on data sets where source code is represented natively as text. Large volumes of source code available online and embedded in technical videos have remained largely unexplored, due in part to the complexity of extraction when code is represented with images. Existing approaches to code extraction and indexing in this environment rely heavily on computationally intense optical character recognition. To improve the ease and efficiency of identifying this embedded code, as well as identifying similar code examples, we develop a deep learning solution based on convolutional neural networks and autoencoders. Focusing on Java for proof of concept, our technique is able to identify the presence of typeset and handwritten source code in thousands of video images with 85.6%-98.6% accuracy based on syntactic and contextual features learned through deep architectures. When combined with traditional approaches, this provides a more scalable basis for video indexing that can be incorporated into existing software search and mining tools.

## CCS CONCEPTS

• **Information systems → Video search**; • **Computing methodologies → Machine learning approaches**; • **Computer systems organization → Neural networks**; • **Software and its engineering → Software libraries and repositories**;

## KEYWORDS

deep learning, convolutional neural networks, video mining, programming tutorials

**ACM Reference Format:**
Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, Erik Linstead. 2018. A Deep Learning Approach to Identifying Source Code in Images and Video. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3196398.3196402


*MSR '18, May 28–29, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196402

## 1 INTRODUCTION

As the amount of publicly available programming resources continues to grow, so too does the volume and diversity of artifacts available to the empirical software engineering community. While substantial progress has been made in the last decade in developing mining techniques for understanding code and those who write it, this research has largely focused on repositories of textual information. This includes source code, user forums, bug tracking systems, and traceability logs, to name only a few application domains. While each of these types of artifacts require their own heuristics to account for differences in form, function, and vocabulary [23], at the lowest level they share a common textual encoding, regardless of whether they represent natural languages, programming languages, or a combination of both. However, a substantial amount of software data also exists in multimedia formats, such as images and video, due to the popularity of online coding tutorials and the relative ease in which they can be recorded and distributed to an Internet-wide audience.

Whether a result of massive open online courses (MOOCs) and similar learning platforms, or companies looking to accelerate the adoption of their products and services by attracting end users, video tutorials have become a staple for software engineers looking to beat either a learning curve or a tricky API into submission. In aggregate, these videos are home to an untold amount of source code, which can be leveraged to extract search features for effective isolation of pertinent video excerpts, or serve as a basis for understanding software development trends in the same way as textual repositories. Yet, because the code is not represented as text, the tools that have worked so well for traditional software corpora are unable to be leveraged on the native data. This typically necessitates the application of optical character recognition (OCR) to make the data compatible with existing techniques. While popular OCR solutions are freely available[35], they are also computationally intensive and sensitive to noise. Thus, if these techniques are to be applied at an Internet scale, performance improvements are necessary.

While there are several viable architectural solutions for identifying source code in video for the purposes of searching or mining, a universal performance bottleneck is found in identifying which video frames contain code, if any, and then processing this code via OCR. This is complicated by the fact that the position of the code within the frame is unknown, and can also change from frame-to-frame (translational variance). This creates substantial overhead as frames must be segmented into candidate regions to identify likely

code snippets, with each region being passed through an OCR algorithm. If no code is found, these processing cycles have been wasted. Tracking unique code examples over the course of a video adds yet another level of cost, as examples appearing across frames must be resolved. Finally, while OCR packages are typically well-tuned for extracting typeset text, handwritten text leads to mixed results depending on the library used. Thus, videos containing handwritten code, on paper or whiteboard in a traditional lecture setting, require additional effort if useful data is to be extracted. Fortunately, advances in computer vision algorithms based on deep artificial neural networks (ANNs) provide a promising direction for computationally inexpensive identification of code-containing frames and the regions within which that code exists.

Deep learning techniques based on artificial neural networks represent the cutting-edge in machine learning in multiple domain applications such as game play[33], biochemical informatics[3, 24], and medicine[21]. Based largely on widely studied methods such as backpropagation and gradient descent, deep learning allows ANNs with deep architectures (multiple hidden layers) to be trained using large volumes of input data. This is made computationally feasible with advances in storage and processing hardware, particularly the availability of a graphics processing units (GPU) and scientific computing libraries [2, 4, 26]. As a result, deep neural networks are well suited for finding high-order, discriminating features in complex data such as images. For this reason, the current state-of-the-art approaches in computer vision are based on deep architectures.

In this paper, we propose a methodology for accelerating code identification and code example resolution in video tutorials by leveraging deep learning. In particular, we apply convolutional neural networks (CNNs) to classify the presence or absence of code in thousands of video frames. Our method achieves almost 93% accuracy on this binary classification task. It is also able to correctly differentiate between typeset code, handwritten code, and partially visible code (for example, overlapping windows obscuring an IDE) with accuracies ranging from 85.6% to 98.6%. The CNNs also provide a natural mechanism for defining regions of interest in the video frame where OCR is most likely to be applicable, thus pruning the space of possible regions that must be examined and thereby increasing the throughput of indexing pipelines.

In addition to code identification with CNNs, we also apply deep autoencoders to the problem of detecting similar or related code examples in video. Autoencoders provide a convenient method for learning compact representations of images. When architected with convolutional layers, they also achieve translational invariance, which can cope with code examples that exhibit position changes across frames. These compact representations allow for substantially faster similarity computation than approaches that rely on pixel-by-pixel comparisons in high-resolution imagery. Our experiments show that our autoencoder approach allows for up to 1.1 million image comparisons per second when run on an appropriate hardware configuration.

The remainder of this paper is organized as follows: Section 2 provides an overview of the dataset used for our experiments, as well as how that data was collected and labeled using crowdsourcing. Section 3 provides a brief description of the deep learning algorithms leveraged in this paper, specifically convolutional neural networks and autoencoders. Section 4 describes the results of our experiments, both in terms of classification accuracy of our models and runtime performance. Related work is described in section 5, with particular attention to CodeTube[29], a previously published code tutorial indexing and search engine that inspired many of the performance optimizations present here. Finally, we discuss conclusions drawn from our work and future directions for computer vision applications in software engineering research in section 6.

## 2 DATA

While neural networks with deep architectures are able to learn high-order features in imagery data, the large number of parameters that are required by these models demand substantial amounts of input data for training. As a first step in exploring the suitability of CNNs and autoencoders for mining source code in video, we cultivated a corpus of 40 tutorials consisting of approximately 22 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 programming language for the experiments described here. A subset of the videos used in this study focused on Microsoft Word, PowerPoint, and other general technology topics. This allowed us to balance our dataset between positive and negative code samples. This diversity in training data allows the network the best chance to recognize code in a variety of scenarios. Though we chose to focus on Java for proof of concept, the methods described here are generic and can be applied to any programming language for which labeled data exists.

All 40 videos were downloaded to our server using pytube[7], a Python library for scraping YouTube videos. After downloading, each video was segmented into a discrete image set by sampling at a rate of one frame per second using FFmpeg[1]. This resulted in 79,500 unlabeled images. A complete list of video URLs from which the frames were extracted, as well as source code used for downloading and processing the videos, can be found at: https://github.com/mlat/msr18.

Upon inspection of the imagery, it was observed that embedded source code was largely presented through four primary mechanisms.

- Visible Typeset Code: Source code is typewritten and completely visible within the frame, such as frames depicting the contents of an IDE, text editor, or PowerPoint slide maximized to fill the screen.
- Partially Visible Typeset Code: Source code is typewritten, but parts of the example are obscured or truncated by other windows or items displayed in the frame.
- Handwritten Code: Source code is written by hand on surfaces such as paper or whiteboards.
- No Code: No code of any type is visible within the frame.

Based on this observation, we decided to use these four categories as the class labels for our machine learning models.

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. Given the tedious and time-intensive nature of manually labeling images, we decided to use a crowdsourcing approach. All images were stored in a relational database, and a web interface was built which would present users with one unlabeled image at a time. The user then selected the label

they believed most accurately described the image, and this was recorded in the database.

To label as many images as accurately and efficiently as possible, help was solicited from approximately 100 students enrolled in a freshman and sophomore-level course focusing on object-oriented programming in Java. Enrolled students represent majors in computer science, software engineering, data analytics, and mathematics. Students were provided with detailed instructions and examples to differentiate among each of the class labels. Of the 100 students contacted, approximately half chose to participate and were given a 3-week window of time for labeling activities. In order to compare our machine learning models to a human baseline, timestamps were recorded every time a label was submitted. Using these estimates, we determined that our human labelers averaged approximately 12 labeled images per minute.

At the end of the labeling period, the database was queried to determine the candidate images for training and testing the performance of our models. Our inclusion criteria limited candidates to images that had been tagged by more than one distinct user. In the case that the labels provided by the users differed, the majority label was taken as the true value. If only two users tagged an image, and they did not agree, the image was excluded from training and testing sets. This process resulted in an image set of 19,200 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, as a final preprocessing step all images were rescaled to 300$x$300 pixels. Though rescaling to this size can result in subtle distortion to the human eye, the reduced resolution allows for greater computational efficiency during the training process. This is especially true for convolutional neural networks, as small filter sizes may be used.

## 3 METHODS

Traditionally, fully-connected, feedforward artificial neural networks (ANNs) receive a single vector representing the raw data at the input layer. This input is then transformed through a series of weighted connections to hidden layers that perform non-linear operations, before being routed to an output layer for the purposes of classification. Each neuron in a layer is connected to every other neuron in the previous layer. The output of each neuron is calculated as a function of the dot product of input values and incoming connection weights. Non-linear relationships in the data are modeled by passing the dot product through an appropriate activation function. This is typically achieved through the application of sigmoidal functions, such as the logistic or hyperbolic tangent function. Because these functions are continuous and differentiable, weights in the network can be trained efficiently using gradient descent to minimize the error between network predictions and truth data through a process known as backpropagation[30]. It can be shown that feedforward neural networks with a single hidden layer and sigmoidal neurons are universal approximators capable of learning any continuous function[5].

Though powerful, structured input data such as images lose their spatial relationships when passed through traditional fully-connected ANN architectures. This is problematic for applications such as computer vision since image features are comprised of groups of pixels. Convolutional Neural Networks (CNNs) represent

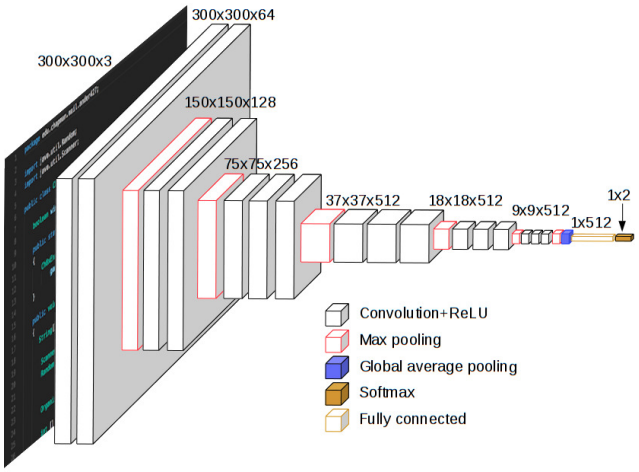


**Figure 1: VGG network, commonly referred to as VGG16 due to its sixteen convolutional layers. Input images are 300x300x3 pixels. The output of the network is a binary classification for the presence of code in the input image. The blue volume in the network represents a global average pooling layer which is required to perform class activation mapping.**

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. The convolutional architecture was inspired by the neurobiological findings of Hubel and Wiesel in experiments performed on the cat visual cortex [14].

Training CNNs with backpropagation was first proposed by LeCunn et al. [20]. 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 in CNNs is typically associated with two primary functions. The first is to reduce the number of free parameters that need to be stored or updated during learning. This can be important in applications where storage space or training data is limited, or where overfitting is a danger. The second function is to apply the exact same operation at different locations in the input data to process the data uniformly and provide a basis for invariance, typically translation invariant recognition in CNN architectures.

To model non-linear data, non-linear activation functions are required. For CNNs, the Rectified Linear Unit (ReLU) [25] is the activation function of choice, as it is computationally inexpensive and avoids the vanishing gradient problem. The ReLU function, shown in equation 1, is a maximum of the input, $x_i$, and 0 to produce the output, $y_i$.

$$y_i = max(0, x_i) \tag{1}$$

A pooling layer is often implemented to downsample the feature space between convolutional blocks. Max pooling is a variant that takes the maximum value in a given window and ignores the rest. Fully connected layers at the end of convolutional networks allow for classification. Another variant, the softmax function, allows for

| **Binary Classification** | Training Set | | Testing Set | |
|---|---|---|---|---|
| | Code | No Code | Code | No Code |
| Visible Typeset Code vs No code | 6,906 | 7,030 | 1,734 | 1,761 |
| Visible Typeset & Partially Visible Typeset Code vs No code | 8,015 | 7,030 | 2,003 | 1,761 |
| Visible & Partially Visible Typeset & Handwritten vs No code | 8,330 | 7,030 | 2,079 | 1,761 |
| Handwritten Code vs No code | 314 | 7,030 | 76 | 1,761 |
| Handwritten Code vs Everything | 314 | 15,045 | 75 | 3,764 |

| **Category Classification** | Training Set | | | | Testing Set | | | |
|---|---|---|---|---|---|---|---|---|
| | VC | PVC | HC | NC | VC | PVC | HC | NC |
| Visible Typeset vs Partially Visible Typeset vs Handwritten vs No Code | 6,906 | 1,109 | 314 | 7,030 | 1,734 | 269 | 75 | 1,761 |

**Table 1: Dataset size for each relative task. The dataset sizes reported in this table were averaged over the five folds in cross-validation. The first five tasks in the table are a binary classification. We predict whether the attribute occurs in an image. In the category classification, we are interested in identifying which of the four categories the image falls into: visible typeset code (VC), Partially visible typeset code (PVC), handwritten code (HC), or no code (NC).**

a probability distribution over classes of interest. Obtaining the $i^{th}$ output of an input vector $z$ is shown in equation 2. Training CNN's is accomplished via a cost function, typically a cross entropy cost in the presence of a softmax. Equation 3 displays the cost function as the negative log probability of the correct answer. Where $t_i$ is the target value of the $i^{th}$ output and $y_i$ is the $i^{th}$ output from the softmax in equation 2. The aim is to maximize the log probability of getting the correct answer.

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2}$$

$$C = -\sum_j t_j \log y_j \tag{3}$$

Convolutional Neural Networks are the current state-of-the-art solution for image recognition tasks. In this paper, we leverage the popular VGG [34] network to label images that contain varying degrees of on-screen code. The VGG network, shown in figure 1, has a convenient architecture in which multiple convolutional operations occur in succession, followed by a max pooling layer for downsampling. Additionally, VGG is shallow compared to networks like Inception [36] and ResNet [10]. This is important because our dataset is small relative to the number of parameters those networks contain. Using a shallower network, such as VGG, allows us to train the network from scratch while avoiding overfitting and achieving a high classification accuracy.

Here we train multiple instances of the VGG network for a variety of classification tasks, described in table 1. The table also provides details on the size of training and testing sets used for each of our experiments. As stated previously, we are interested in predicting if an image frame contains typeset code, partially visible typeset code, handwritten code, or no code at all. The output of the network, for the first five cases listed in table 1, is a binary classification. The last case represents a multiclass classification problem where we are interested in predicting membership across all four of these categories.

Within software engineering tutorials, a single frame may sit idle for a given time interval while the instructor narrates over the frame. As our data is randomly shuffled to produce training and testing sets, this may lead to duplicates appearing in both sets and result in overfitting. To avoid this, a standard pixel-wise comparison is implemented to remove frames from the test set if they appear too similar to frames in the training set. To further assess the generalizability of our models, our experiments were performed using five-fold cross-validation. At the start of each fold, network weights were reinitialized to create a new network. Simulations were 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. Individually these GPUs are capable of achieving 4.7 TeraFLOPs and 9.3 TeraFLOPs of single and double-precision floating point performance, respectively.

Class Activation Mapping (CAM) gives convolutional networks tremendous localization ability despite being trained on image-level labels [42]. CAM requires the use of a global average pooling layer [22], which is added to the last convolutional layer of the VGG network. The Keras Visualization Toolkit [19] is used to produce CAM results. 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 code and not other circumstantial features contained in the images. Additionally, leveraging the CAM results allows us to quickly identify appropriate image regions that can be passed to an OCR library in order to tokenize and index contained code for further search or mining activities.

During a tutorial, a code example may appear at various points throughout the video or be built up incrementally over time. Thus, it is convenient to be able to identify which frames are likely related to the same code example. This process can be computationally intense, however, as it requires a substantial number of pixel-by-pixel comparisons across all pairs of frames. To speed up this process, we apply autoencoders. An autoencoder is a type of neural network that is trained to reconstruct its input. Unlike feedforward and convolutional neural networks, autoencoders are unsupervised.

They experience features of the data but are not dependent on truth data. Instead, an autoencoder is trained to minimize the distance between its input and its output.

Formally, an autoencoder network may be viewed as consisting of two parts: an encoder function, $z = f(x)$ , and a decoder function that produces a reconstruction of the input, $r = g(z)$ [8]. Internally, it has a hidden layer, $z$ , that represents a compact encoding of the original data. Because the model is forced to compress the data into a compact representation and then reconstruct the original input, the network learns to extract the most relevant features for encoding. Autoencoders have been shown to be comparable to principle component analysis [12]. The concept of autoencoders can be applied to images by using convolutional layers. This allows for images to be encoded into a compact representation, maintaining the most important features in the encoding. Images can then be compared against each other in a lower-dimensional space using any appropriate distance metric (here we use Euclidean distance), requiring fewer operations and less CPU time.

In order to facilitate reproducibility of our experiments, all Python source code used for training, testing, and validating the performance of our CNNs and autoencoders is available from the GitHub link provided in section 2.

## 4 RESULTS

Here we detail the results of applying our deep learning architectures to code image classification and similarity analysis.

### 4.1 CNN Experiments

To begin, convolutional neural networks were trained using 5-fold cross-validation for the 5 binary classification tasks and 1 multiclass classification task described in table 1. Each convolutional model took, on average, 2.5 hours per fold to train, for a total of 62.5 computing hours of training for all folds in all models. In practice, overall time was decreased by training models in parallel by taking advantage of multiple GPUs on our deep learning server. Though computationally intensive, training is done a priori, and cost can be amortized over a model's lifetime once deployed to automatically classify images.

Table 2 displays the mean and median accuracies of the 5-fold cross-validation experiments for each classifier. An accuracy of 92.92% is achieved on the binary classification task of predicting the presence of visible typeset code. Combing the visible typeset and partially visible typeset code categories yields an accuracy of 90.3% when predicting the presence of typeset code. Binary classification on the amalgamation of code categories (visible, partially visible, and handwritten) produces an accuracy of 90.52%. When predicting the probability distribution over each of the four categories an accuracy of 85.59% is achieved. The confusion matrix for predicting one of four categories is shown in figure 2. The confusion matrix reports that 129 images were incorrectly predicted as visible code when the ground truth was partially visible. This error is understandable, as even human labelers have a difficult time choosing the correct category. In certain cases, we observed the same human labeler to mark a given image as visible code one time and partially visible code the next.

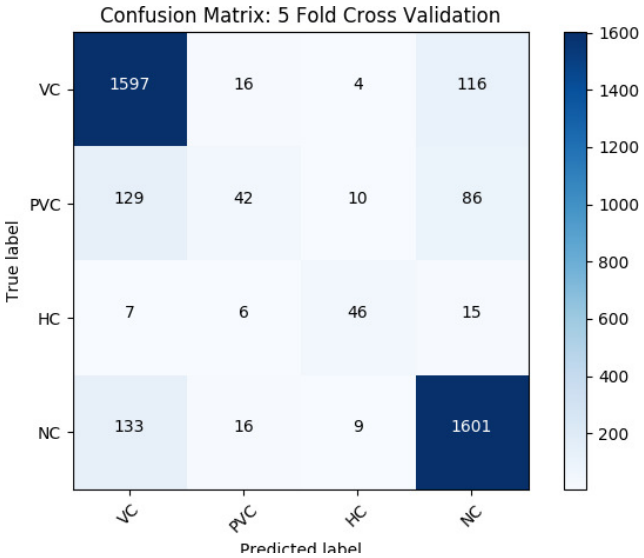


**Figure 2: Confusion matrix for the task of predicting one of four categories: visible typeset code (VC), Partially visible typeset code (PVC), handwritten code (HC), or no code (NC).**

Two separate experiments were conducted on handwritten code detection. The first was to predict handwritten code versus no code at all and the VGG network achieved an average accuracy of 98.191%. The second task was to predict handwritten code versus both visible and partially visible typeset code, and an accuracy of 98.6% was found. The latter task is slightly more difficult, as the network must learn to disambiguate common syntactical features that occur in both handwritten and typeset code (indentation structure, semicolons, curly brackets, etc.).

In regards to handwritten code, we note two possible limitations to our results. The first is that the proportion of the dataset that falls into this category is very small relative to the non-handwritten portion. On the two handwritten code detection tasks described above, accuracies of 95.7% and 98.01% respectively were achievable if the network was only to predict no code. However, as the networks in these tasks achieved a five-fold cross-validation mean of 98.2% and 98.6% respectively, it is clear the networks have learned some distinguishing features between handwritten code and non-handwritten code.

This brings us to our second concern regarding handwritten code detection. The network tends to learn features associated with handwritten code, but not directly related to it. For example, the network tends to predict an image of a man standing in front of a whiteboard as containing handwritten code. This is ostensibly due to the fact that in software engineering tutorial videos, handwritten code is often illegible, even to human viewers. Thus, black markings on a whiteboard in the presence of other common features, like a person, will cause the network to strongly predict the presence of code. However, while the results on handwritten code are promising, further experimentation with a larger dataset is needed.

| | Mean Accuracy | Median Accuracy | Precision | Recall |
|---|---|---|---|---|
| Visible Typeset Code vs No code | 92.917 | 92.725 | 0.939 | 0.919 |
| Visible Typeset & Partially Visible Typeset Code vs No code | 90.323 | 90.182 | 0.905 | 0.890 |
| Handwritten Code vs Everything | 98.609 | 98.646 | 0.991 | 0.995 |
| Handwritten Code vs No code | 98.191 | 98.162 | 0.987 | 0.994 |
| Visible & Partially Visible Typeset & Handwritten vs No code | 90.518 | 90.499 | 0.903 | 0.884 |
| Visible Typeset vs Partially Visible Typeset vs Handwritten vs No Code | 85.594 | 85.547 | 0.839 | 0.856 |

**Table 2: Mean and median accuracies as well as precision and recall scores, of the five-fold cross-validation experiments.**

Equally as promising as the classification accuracies of the models is the speed in which they are able to label images. Once the networks have been trained, using them in practice is highly efficient. We found that the VGG networks can label 4,500 images per minute on average using the GPU. This represents a 375x improvement over our human baseline. In practical terms, this means an hour-long video can be completely labeled in under 50 seconds, compared to the 300 minutes required by a human, assuming a sampling rate of 1 frame per second.

Figure 3 shows CAM results on correctly predicted Java code image frames. 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 3, the left column shows examples from the test set, while the right column shows the CAM overlaid on the corresponding test image. Visual analysis of the CAM result images reveals the network's preference towards features such as method and class declarations, semicolons, and curly brackets. Additionally, contextual features like keywords, package imports, and variable initializations show up favorably in CAM results. The first row of figure 3 shows an image of handwritten Java code, correctly identified by the network. The CAM image on the right shows that the network achieves this identification through recognition of the class declaration, method heading, and curly brackets. The second through fourth rows show examples of typeset Java code tagged correctly by the network. In cases such as the last row of figure 3, the CAM heatmap covers nearly the entire body of the image. This is a consistent occurrence when the predominant focus of the image is Java code and code features are easily visible. The results of figure 3 show the network is capable of learning syntactic and contextual features of image code frames.

In addition to providing a convenient mechanism for visualizing learned features, the heat maps also provide a heuristic for identifying regions for OCR. In particular, "hot" areas can be thought of as the center point for candidate bounding boxes. The length and height of the bounding rectangles can be adjusted by examining the distribution of "heat" over a region.

Though our experiments here have focused solely on the identification and tagging of video frames that contain the Java programming language, the ability of the models to discern between Java and other languages is of practical significance. Thus, we presented our trained CNNs with images of programming languages other than Java to see how well they faired. These tests were met with varying degrees of accuracy. Figure 4 shows examples of the network on video frames of C and HTML respectively. Java and C share many syntactic attributes such as curly braces, semicolons, and method declarations (in a general sense). This leads the network to be easily fooled by these code snippets. However in the case of HTML, Java differs significantly. The bottom row of figure 4 shows that the network is able to differentiate HTML from Java by identifying the presence of many angle brackets, a feature that is not common to Java.

The Python programming language is somewhat of a middle ground, as it's syntax is not as far from Java as HTML and not as similar as C. The results of the network classifying Python code snippets can be seen in figure 4. The first Python example is correctly identified as non-Java code. From the CAM results, it appears that the network identifies the absence of semicolons and curly braces. However, the second Python example is incorrectly classified. One possible reason for this is that the text in this image is smaller, resulting in lower quality. This makes it harder for the network to identify key features such as semicolons and curly braces that are harder to see. As a result, the network focuses on the main body and shape of the indented code.

It is important to emphasize that our models are at a disadvantage when asked to differentiate Java from other programming languages because no labeled examples of other languages were provided to the CNNs during training. Thus, our tests on these tasks do not represent robust experiments, but only initial attempts to understand what Java-specific language features were learned by the networks. To obtain more precise results would require the hand labeling of datasets for each language. Acquiring more labeled data samples of other languages for training will allow the network to better generalize to languages other than Java. This work is currently in progress.

## 4.2 Autoencoder Experiments

Having demonstrated the efficacy of CNNs on identifying code in video, we turn to the task of determining which code samples within a tutorial represent the same example. This is complicated by the fact that even identical code samples across frames are subject to positional variation due to vertical and horizontal translations caused by scrolling or window repositioning. Additionally, a code example may be built up incrementally over frames, resulting in images that are similar but not identical.

To automate the process of similarity analysis we trained an autoencoder with convolutional layers on images labeled as containing code. This process took a total of 15 hours on a single GPU. Once the autoencoder was trained, all images were transformed into the encoder's learned compact representation. We observed a rate of approximately 20,000 images per minute for this task (less than one second for an hour-long video), which was completed in parallel using both GPUs in our server.

To test the ability of the autoencoder to efficiently identify related code examples, a small group of human experts selected excerpts from videos not used during training that exhibited repetition of identical or similar code examples across frames. This resulted in 169 frames containing code as determined by our CNN approach. These frames were passed through the autoencoder for compression, and the resulting vectors were clustered using Euclidean distance with NumPy[27]. The cluster memberships were then examined manually and the approach was found to achieve 85.7% accuracy for this experiment. In terms of runtime performance, by using the autoencoder to compress the video frames, we were able to achieve 1.1 million frame comparisons per minute. This is a 7.6x speedup over brute force, pixel-by-pixel image comparison which achieved a maximum of 145,000 comparisons per minute on our hardware. Additionally, because we leveraged convolutional layers in our autoencoders, our comparisons were able to account for some degree of translational variance in our images.

While our results are promising, it should be noted that a limitation of this experiment is the relatively small number of testing examples used to determine accuracy. We are currently curating a much larger synthetic dataset that uses a series of seed code example images, and then randomly translates these images throughout the pixel space. Additionally, more truth data is needed for the case where an example is built up iteratively over several frames. Once we have acquired this data we can more aggressively tune the parameters of the autoencoder to maximize generalizability and accuracy.

## 5 RELATED WORKS

In recent years the study of software repositories has been notably shaped by deep learning techniques that have consistently improved upon existing knowledge and systems. The applicability of deep learning to software repositories is presented in [38]. As the authors demonstrate, deep learning algorithms have led to impressive advances in fields like natural language processing (NLP), and thus should be looked to when it comes to advancing the study of software corpora. Specifically, they show how deep learning can be used to model sequential data, aid in generalization, and optimize real SE tasks like code suggestion.

In [32] the authors train two Long Short Term Memory (LSTM) networks to pinpoint the location of syntax errors. In [9], the authors outline DeepAPI, an adapted neural language model (specifically a Recursive Neural Network Encoder-Decoder) that generates API sequences given a natural language query outperforming existing bags-of-words approaches.

Deep learning methods have also been applied to advance clone detection in a variety of mediums. In [15], a framework referred to as CDLH is proposed which applies LSTM networks to compare code representations using hamming distance. This can be compared to the Deckard framework outlined in [18] which proposes a clone detection tool based on syntax analysis. In [37], a language model is leveraged to convert lexical elements of code to continuous valued vectors referred to as embeddings. These embeddings are then compressed through the recursive application of the autoencoder. This compression allows for efficient comparison of code fragments to propose potential functionality repetition found in source code repositories. In our paper we also leverage the autoencoders ability to compress large entities by compressing matrices of pixels in order to compare frames efficiently.

The application of other deep learning techniques, specifically CNNs, has further contributed to a widening of research avenues. CNNs are used to recognize handwritten English characters in an offline setting in [41]. Further, in [39] the authors utilize CNNs to build a multiclass classification system in order to link knowledge units like a question and its answer in Stack Overflow. This multiclass, deep learning based system outperforms existing traditional, human-engineered classifier systems which assume a binary relatedness (i.e. related or not) and ultimately fail when encountering lexical gaps.

CNNs have also been used extensively in the field of digital image processing because of their powerful and accurate object recognition abilities. CNNs have been shown to exhibit particular prowess in the area of text detection and recognition when used with OCR algorithms, and have been used in video-tagging applications for robust scene-text recognition in multiple different languages. However, to the best of our knowledge, CNNs have yet to be applied towards identifying source code in software engineering tutorial videos. There have been several successful endeavors in recent years to tag and extract relevant portions of programming tutorial videos based on code detection, but these approaches have utilized standard image processing techniques and not deep learning.

### 5.1 Text Detection

In [13], a coarse-to-fine strategy is used to extract text from video without constraints. The authors employ a layered CNN approach that first utilizes the networks to generate candidate text regions and then looks to enable feature sharing and identify final text regions after projection analysis. Their approach results in a robust multimedia indexing and retrieval system. In [11] the authors continue in their explorations of deep learning through another text-focused CNN. Particularly, they center on extracting features related to text from images by training their CNN with multi-level supervised information. They propose training this novel network with text region mask, character label, and binary text/non-text information. We followed a similar training strategy by hand-labeling our images to indicate the presence of code contained in the video frame.

In [16], the authors present an approach that does not rely on hand labeling. They train large CNNs to perform word recognition on entire proposal regions (as opposed to individual characters) at one time. In this study, they detail how they assembled a pipeline for large-scale detection of text in video. These authors have also employed CNNs in character classification, as outlined in [17]. Their

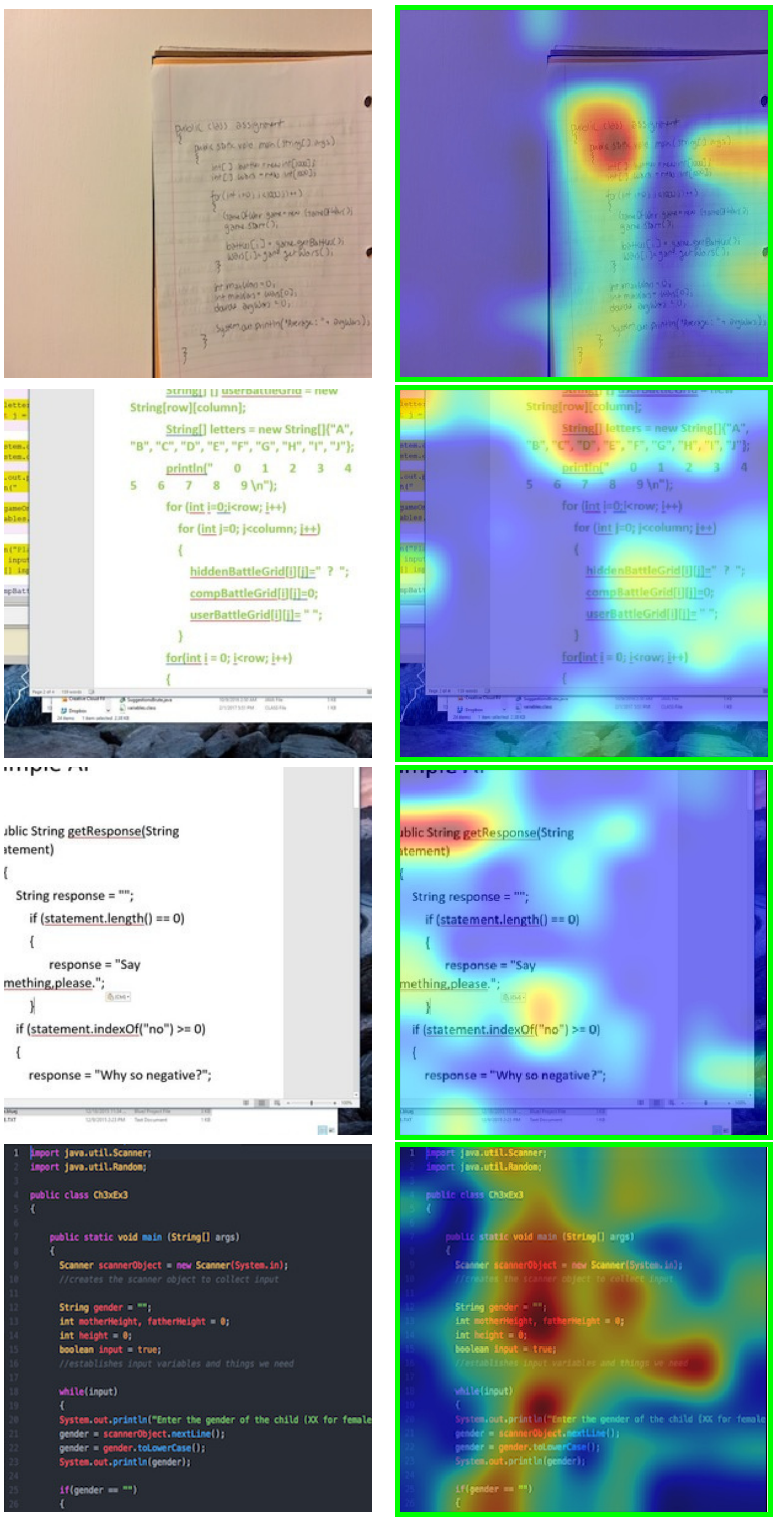

**Figure 3: CAM results on correctly (green border) predicted Java code image frames. Normal test image (left column). CAM results on the test image (right column). The first row shows an image of handwritten Java code. The second through fourth row shows examples of typeset Java code.**

**Figure 4: CAM results on languages other than Java. A green border means the image was correctly predicted as not containing Java code. A red border means an incorrect prediction. The top row shows an example of C code. The second row shows HTML code. The bottom two rows are Python code.**

approach in that endeavor computed text saliency maps by evaluating a character vs. background CNN classifier in a sliding window fashion. Bounding boxes on the text were then derived using these text saliency maps. Similarly, in our approach, we utilize class activation mapping to project the output category of our CNN back onto the input image, generating a heat map of the zones of interest where the network predicts code to appear. This way, subregions of the image can be identified in one simple step, using work already done by the network itself.

The task of identifying code in video highlights a need for a robust model to detect scene text in video frames. In [31], a CNN-based architecture for the automatic recognition of color text characters extracted from scene images is proposed. The authors utilize seven heterogeneous convolution layers and combine the automatically learned operations to extract features with strong generalizability, allowing uppercase and lowercase versions of each letter to be represented by the same class. Their approach is robust to distortions, complex background, low resolution, and non-uniform lighting, a crucial characteristic in the indexing of tutorial videos, which vary greatly in quality and presentation.

### 5.2 Code Detection

Video indexing based on recognition of code in captured frames is an emerging field of study given the growing volume of online code tutorial videos. However, this problem has yet to be approached from a deep learning perspective. Rather, researchers in the field have found success with more standard image processing methods.

One such notable approach to digital image processing in the code recognition domain is outlined in [29]. Throughout this study, the authors implement a tool created to enable developers to access a greater breadth of expertise by allowing users to query the contents of video tutorials. The methodological foundations of this paper largely served as a motivation for our research and informed the decisions we made in the building of our model. The authors first optimized their solution by comparing the pixel matrices of consecutive video frames. If they differed by less than 10% only the first frame was kept. This highlighted the importance of removing identical frames and in our paper, we have leveraged the abilities of the autoencoder in order to further optimize frame similarity identification.

The authors go on to identify Java code by applying OCR to a candidate sub frame followed by the application of a parser which creates a Heterogeneous Abstract Syntax Tree. This construction is then used to conclude if code is present in the frame. This approach informed our research as we saw an opportunity to further the advancements made in the CodeTube study through using CNNs to identify when code was present in the frame to allow for a more specific application of OCR.

The authors addressed other pertinent issues in parsing software video tutorials such as code that appears in multiple frames but translated to different areas in the frame. To remedy this, the authors compare the Java constructs generated by OCR applications and the parser outlined above to identify if consecutive code-containing frames contain the same code component. In order to improve the accuracy of this comparison if these constructs are not found to be similar, Longest Common Substring (LCS) analysis is run on the pixel matrices in order to determine similarity. LCS is very effective in this environment as it is not affected by scrolling that is often conducted in video tutorials however, the expense of its application motivated us to utilize an autoencoder based approach. Our method achieves translational invariance and is thus able to address the issue of code samples changing positions on a screen while also improving on the efficiency of similarity computation. It should be noted that the authors expanded their solution beyond parsing video tutorials by integrating other sources of expertise, particularly Stack Overflow discussions, and conducted a thorough user study.

Another approach to indexing programming tutorials was conducted in [40]. The authors base their work on a general application of OCR to video tutorials to consolidate code as it appears across frames followed by the leveraging of programming language statistical models to determine if the extracted text is code and then, to correct the code downloaded in order to produce higher quality results. Finally, in [6] the authors describe a method of tagging programming videos based on title, description, and audio transcript. This project did not use machine learning or image recognition to consider the visuals of the video data but pulled from those text-based sources to identify relevant topics.

## 6 CONCLUSION AND FUTURE WORK

In this paper we have described the application of deep learning techniques, specifically convolutional neural networks and autoencoders, to the task of identifying source code examples in video frames from a large dataset, and examining those examples for similarity. Though CNNs represent the state-of-the-art in computer vision, we believe this work represents their first application to mining software from multimedia data sources. Our results demonstrate 85.6% to 98.6% accuracy on classification tasks and is capable of generalizing to typeset, partially visible, and handwritten code examples. We are also able to achieve 85.7% accuracy for similarity prediction, albeit on a much smaller test sample. Additionally, our approach yields substantial runtime performance increases when used in conjunction with GPU-based computing platforms and appropriate scientific computing libraries.

We are currently working on curating additional labeled data for a variety of programming languages, including C++, and R and have begun an initial exploration into differentiating between Python and Java code samples embedded in digital images through a model that can differentiate between multiple languages while learning lexical features in the process [28]. Using this data we will train an ensemble of classifiers for identifying these languages in video and images. We are also finalizing a web API that can be used by users to submit the URL of a tutorial video and receive in return an indexed list of frames that contain code, as well as groups of frames corresponding to the same or similar code examples. We hope that this will help further the development of code tutorial indexing platforms, such as the ones discussed in our related works section. Finally, in the longer term, we are interested to see if computer vision techniques can be used to detect poorly implemented code examples of well-known algorithms.

## REFERENCES


[1] [n. d.]. FFmpeg. ([n. d.]). https://www.ffmpeg.org/

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[3] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. 2015. Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning. *Nature biotechnology* 33, 8 (2015), 831–838.

[4] François Chollet et al. 2015. Keras. https://github.com/keras-team/keras. (2015).

[5] George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2, 4 (1989), 303–314.

[6] Javier Escobar-Avila, Esteban Parra, and Sonia Haiduc. 2017. Text retrieval-based tagging of software engineering video tutorials. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 341–343.

[7] Nick Ficano. 2018. nficano/pytube. (Jan 2018). https://github.com/nficano/pytube

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[9] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[11] Tong He, Weilin Huang, Yu Qiao, and Jian Yao. 2016. Text-attentional convolutional neural network for scene text detection. *IEEE transactions on image processing* 25, 6 (2016), 2529–2541.

[12] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

[13] Ping Hu, Weiqiang Wang, and Ke Lu. 2015. Video text detection with text edges and convolutional neural network. In *Pattern Recognition (ACPR), 2015 3rd IAPR Asian Conference on*. IEEE, 675–679.

[14] David H Hubel and Torsten N Wiesel. 1962. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology* 160, 1 (1962), 106–154.

[15] Ming Li Huihui Wei. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[16] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2016. Reading text in the wild with convolutional neural networks. *International Journal of Computer Vision* 116, 1 (2016), 1–20.

[17] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Deep features for text spotting. In *European conference on computer vision*. Springer, 512–528.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[19] Raghavendra Kotikalapudi and contributors. 2017. keras-vis. https://github.com/raghakot/keras-vis. (2017).

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

[21] Christine Lee, Ira Hofer, Maxime Cannesson, and Pierre Baldi. 2017. Deep Learning for Predicting in Hospital Mortality. In *ANESTHESIA AND ANALGESIA*, Vol. 124. LIPPINCOTT WILLIAMS & WILKINS TWO COMMERCE SQ, 2001 MARKET ST, PHILADELPHIA, PA 19103 USA, 85–86.

[22] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[23] 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.

[24] Alessandro Lusci, Gianluca Pollastri, and Pierre Baldi. 2013. Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. *Journal of chemical information and modeling* 53, 7 (2013), 1563–1575.

[25] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[26] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (2008), 40–53.

[27] Travis E. Oliphant. 2001–. Numpy: Python for Scientific Computing. (2001–). http://www.scipy.org/

[28] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning Lexical Features of Programming Languages from Imagery Using Convolutional Neural Networks. In *2018 IEEE/ACM 15th International Conference on Program Comprehension (ICPC)*.

[29] 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.

[30] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533.

[31] Zohra Saidane and Christophe Garcia. 2007. Automatic scene text recognition using a convolutional neural network. In *Workshop on Camera-Based Document Analysis and Recognition*, Vol. 1.

[32] Eddie A Santos, Joshua C Campbell, Abram Hindle, and JosÃĺ Nelson Amaral. 2017. Finding and correcting syntax errors using recurrent neural networks. *PeerJ Preprints* 5 (Aug. 2017), e3123v1. https://doi.org/10.7287/peerj.preprints.3123v1

[33] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan. 2016), 484–489. https://doi.org/10.1038/nature16961

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

[35] R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02 (ICDAR '07)*. IEEE Computer Society, Washington, DC, USA, 629–633. http://dl.acm.org/citation.cfm?id=1304596.1304846

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[37] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[38] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 334–345. https://doi.org/10.1109/MSR.2015.38

[39] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2970276.2970357

[40] 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.

[41] A. Yuan, G. Bai, L. Jiao, and Y. Liu. 2012. Offline handwritten English character recognition based on convolutional neural network. In *2012 10th IAPR International Workshop on Document Analysis Systems*. 125–129. https://doi.org/10.1109/DAS.2012.61

[42] 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.