Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning

Mohammad Alahmadi, Jonathan Hassel, Biswas Parajuli, Sonia Haiduc, Piyush Kumar
Department of Computer Science - Florida State University
Tallahassee, FL, USA
{alahmadi, hassel, parajuli, shaiduc,piyush}@cs.fsu.edu

ABSTRACT

Background: Video programming tutorials are becoming a popular resource for developers looking for quick answers to a specific programming problem or trying to learn a programming topic in more depth. Since the most important source of information for developers in many such videos is source code, it is important to be able to accurately extract this code from the screen, such that developers can easily integrate it into their programs. *Aims*: Our main goal is to facilitate the accurate and noise-free extraction of code appearing in programming video tutorials. In particular, in this paper we aim to accurately predict the location of source code in video frames. This will allow for the dramatic reduction of noise when using extraction techniques such as Optical Character Recognition, which could otherwise extract a large amount of irrelevant text (e.g., text found in menu items, package hierarchy, etc.). *Method*: We propose an approach using a deep Convolutional Neural Network (CNN) to predict the bounding box of fully-visible code sections in video frames. To evaluate our approach, we collected a set of 150 Java programming tutorials, having more than 82K frames in total. A sample of 4,000 frames from these videos were then manually annotated with the code bounding box location and used as the ground truth in an experiment evaluating our approach. *Results*: The results of the evaluation show that our approach is able to successfully predict the code bounding box in a given frame with 92% accuracy. *Conclusions*: Our CNN-based approach is able to accurately predict the location of source code within the frames of programming video tutorials.

CCS CONCEPTS

• **Software and its engineering** → *Documentation*; • **Computer vision** → *Image recognition*; • **Computer systems organization** → *Neural networks*;

PROMISE'18, October 10, 2018, Oulu, Finland

ACM ISBN 978-1-4503-6593-2/18/10…$15.00
https://doi.org/10.1145/3273934.3273935

KEYWORDS

Programming video tutorials, Software documentation, Source code, Deep learning, Video mining

ACM Reference Format:
Mohammad Alahmadi, Jonathan Hassel, Biswas Parajuli, Sonia Haiduc, Piyush Kumar Department of Computer Science - Florida State University and Tallahassee, FL, USA {alahmadi, hassel, parajuli, shaiduc,piyush}@cs.fsu.edu [3.0ex]. 2018. Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning [1.0ex]. In *The 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'18), October 10, 2018, Oulu, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3273934.3273935

1 INTRODUCTION

Nowadays developers spend 20-30% of their time online, looking for information they need for their daily development and maintenance tasks [2, 6]. Most of this time is spent consulting online documentation in the form of Q&A websites, tutorials, API documentation, etc. Programming video tutorials are one documentation source that has seen a rapid growth in production and usage [13], though some limitations still prevent developers to use them to their full potential. In particular, given that copy-pasting code is the most common task developers perform online [2], video tutorials are currently not very helpful in that regard. Code appearing on the screen is often not available for download or copy-pasting, which makes videos often inconvenient to use as a resource. Therefore, designing tools and techniques that can automatically extract correct code appearing in video tutorials is of extreme importance, as it would give developers access to a wealth of documented source code currently not leveraged.

Recent approaches have started moving in this direction [20, 24] by using Optical Character Recognition (OCR) techniques in order to extract the code found in software development video tutorials. However, when used on a video frame containing source code, OCR will not only extract the code, but every piece of text that is appearing on the screen, including text found in menu items, file and package hierarchies in IDEs, desktop icons, error messages, the program output, etc. Given that OCR reads text line by line, it often mixes the non-code text with the source code [23], resulting in noise and incorrect, unusable code being extracted. Therefore, it is necessary to first accurately identify the section of the screen where the code is located and then apply OCR only to that section, in order to eliminate the noise.

Current approaches [20, 24] have made use of heuristics and assumptions in order to determine the location of the code on

the screen, such as generalizing its location based on only one video [24], or setting hard thresholds and dividing images into predefined fixed-size sub-images [20]. However, these heuristics are not generalizable and fail to identify the location of the code on the screen when videos do not follow the assumptions made.

In this paper we propose a novel, robust approach based on object identification and deep learning algorithms to predict the presence and precise location of source code in the frames of a software development video tutorial, which is the first important step towards the extraction of correct code from videos. Our approach starts off by segmenting a programming video, obtaining one static frame for every second of footage in the video. Then, given that many of the frames of a programming screencast contain the same image [3], it detects and removes redundant or duplicate frames using object identification algorithms. Finally, for each of the remaining frames, the approach uses a Convolutional Neural Network (CNN) to predict the bounding box surrounding any code fragment found in the frame (or zero if the frame does not contain code).

We evaluated our approach on a set of frames extracted from 150 Java video programming tutorials hosted on YouTube[1]. The videos initially contained more than 82K frames in total, with 50K remaining after the redundant frame reduction step. From these, a sample of 4,000 frames were manually annotated with their code bounding boxes and used as ground truth in our evaluation experiment. The results of the experiment show that our approach is able to accurately determine the area of the screen where the code is located, achieving an accuracy of 92% on the used dataset. We make our dataset, scripts, and results available in our *replication package*[2] (17GB in total).

The rest of the paper is organized as follows: Section 2 presents some background information on deep learning for image analysis, Section 3 introduces our approach and its components, and Section 4 describes our evaluation on predicting the location of code in Java video programming tutorials. Further, Section 5 presents current limitations of our approach, Section 6 discusses threats to the validity of our results, Section 7 presents the related work and finally Section 8 concludes the paper.

2 DEEP LEARNING FOR IMAGE ANALYSIS

In our approach for determining the location of code in video frames we employ deep learning techniques, and in particular Convolutional Neural Networks (CNN), which are a type of Artificial Neural Networks (ANN) that have been shown to perform the best for image analysis and classification [8]. In this section we give a brief introduction to deep learning and the networks we use.

An ANN is a collection of connected nodes called neurons [5]. These neuron nodes are interconnected much like neurons are in the human brain. Each neuron is characterized by its weight, bias, and activation function. Each connection is initially assigned a random weight which increases or decreases the strength of the signal at a connection. Each neuron node is responsible for processing a signal sent to it, then sending the processed signal to all of the other neurons connected to it. This signal is a number, and the output of each neuron node is calculated by a non-linear function of the sum of its inputs.

An ANN is composed of different layers of neural nodes: an input layer, a number of hidden layers, and an output layer [5]. Each node in a layer is connected to every node in the previous layer. The neuron nodes in the initial hidden layer are fed input from the input layer. They then process this input and send a signal forward through all of their connections. This continues until the signal reaches the output layer. As learning in the network progresses, the weights are re-calibrated during a backpropagation phase based on errors that were found in the output [22]. This increases the accuracy of the predictions that the ANN makes.

A Convolutional Neural Network (CNN) [9], also known as a ConvNet is a type of ANN that has proven to be very effective when it comes to image recognition and classification [8]. CNNs are particularly good at detecting patterns in images through the use of filters. The filters slide or "convolve" across the image until the entire image is covered. These filters can detect things like edges, shapes, textures, etc. and where they are located in an image.

The *convolutional layer* is what separates a CNN from other ANNs and is where most of the computation happens. In a CNN the convolutional layer is typically followed by a non-linear layer and sometimes a pooling layer. At the end of the network there is a fully connected layer to equip the network with the ability to classify samples. So, in general there are four main operations in a CNN architecture: convolution, non-linearity, pooling, and classification.

Convolution: A convolutional layer consists of a set of filters, also known as kernels or feature detectors. A filter is a set of weights which are learned using backpropagation. These filters are used to extract features from the image. The first layers in a CNN can detect large features while later layers can detect smaller and more abstract features. The filters' size is set as one of the parameters in a convolution layer. Every image can be considered as a matrix of pixel values, and every filter is a matrix of weighted values. Features are extracted by sliding the filter over the image pixel by pixel, and at every stop, or stride, computing the element-wise multiplication between the two matrices, then summing all the multiplication outputs to get the single element for the output matrix, or feature map. The size of the feature map is controlled by three parameters: depth, stride and zero padding. The depth is determined by the amount of filters used in the convolutional layer. For example, if eight filters are used then there are eight different feature maps. The stride is the number of pixels that the filter is sliding over in the original image after each dot product is computed. A larger stride produces a smaller feature map. Zero padding is when the input matrix is padded with zeros around the borders so that features can be extracted from the border of the images. Every neuron in a convolutional layer is connected to only a small region of the input volume, i.e., the region that the filter is covering, which is also referred to as the receptive field.

Non-Linearity: A Rectified Linear Unit (ReLU) layer is used for non-linearity. It uses an activation function to define the output of each node [14]. The ReLU replaces all negative pixel values in a feature map with a 0. This helps to decide whether the information the neuron is receiving is relevant or should be ignored.

Pooling: The pooling layer, also referred to as the downsampling layer reduces the dimensionality of each feature map, but keeps

[1] https://youtube.com/
[2] https://goo.gl/R7yGiD

the most important information. This layer typically takes a filter of size 2x2 with a stride of the same size, and convolves it across the feature map, performing calculations after each stride. The feature map can be downsampled in many different ways. One of the most popularly used downsampling techniques is max pooling [5]. When using max pooling, the 2x2 filter takes the maximum value that it is covering on the feature map and then it slides to the next region by its stride distance, where it performs this calculation again until it has covered the entire image. The resulting feature map is dimensionally reduced. This is an important step because it reduces the number of parameters or weights, which then lessens the computational cost. It also helps control overfitting, which is when a model gets so tuned to the training examples that it is not able to predict well on testing data.

Classification: The fully connected layers are placed at the end of the network, and give the network the ability to classify samples. These layers are called fully connected because every neuron in these layers is connected to every other neuron in the previous layer. The output of the ReLU and pooling layers represent features of the input image. The purpose of the fully connected layer is to use these features to classify the input image into various classes based on the training data. The fully connected layer uses a softmax activation function in the output layer to give a probability that the input image is one of the various classes. This softmax function takes a vector of real-values and normalizes it into a vector of values between 0 and 1 that sum up to one. The value with the greatest magnitude in this vector represents the prediction of the model.

3 APPROACH

This section presents the two steps of our approach in detail, namely the redundant frame removal step and the detection of the code bounding box.

3.1 Redundant Frame Removal

Recent work [3] found that programming screencasts are much more static than other types of videos. In other words, the footage remains fixed on the same images for longer periods of time. This is especially true when the tutors in the videos start coding on the fly. In these video fragments, the tutor first writes some code and then stops and explains it while the image remains static on the code editor. Moreover, some of these videos do not only contain source code, but also material presented on slides, in API documentation, on a board, etc. where tutors spend some time explaining the static material shown on screen. This means that when segmenting videos in frames extracted every few seconds, as it is customary when analyzing programming tutorials [3, 15, 18–20, 24], this results in many redundant images as a result of a frame being idle for a while.

Analyzing video frames is computationally intensive and the evaluation of these techniques often involves some form of manual labeling by participants [15], which is time consuming. Therefore, reducing the amount of redundant information analyzed is a very important step towards scalability. While the frames could be sampled at longer periods of time to reduce redundancy, this would inevitably lead to the loss of important information, as some of the frames skipped may contain new material. Our approach employs a novel technique that aims not only at reducing redundant frames, but also at ensuring that new information is not omitted.

We first extract frames for each video at every second. Then, the first step towards identifying and removing redundant frames is detecting the similarity between neighboring frames. A variety of algorithms for computing a similarity value between two images can be employed. Previous work on analyzing programming video tutorials has made use of pixel-by-pixel metrics [3, 19, 20] to compute the similarity between frames. This however, has two main disadvantages. First, it is very computationally intensive. For example, comparing two 1080p HD frames pixel-by-pixel requires a huge number of comparisons as each frame contains more than two million pixels. Second, the comparison can be inaccurate, since two images containing the same information, but having different scales, rotation, quality, etc. can be considered as different images. This is a limitation acknowledged in previous work [18] and is important to consider as we noticed that in some of the videos the tutors zoomed-in while explaining a code snippet.

In this paper we leverage a faster and more robust algorithm called Scale-Invariant Feature Transform (SIFT) for determining the similarity between neighboring frames. SIFT compares frames based on their *features* rather than their individual pixels and has been successfully used in image matching [11, 12]. SIFT is based on identifying key-points in the images, which are important areas that are used as discriminating points and are the basis for the comparison between two images. SIFT's algorithm has four main steps: approximating key-point locations, refining key-point locations, assigning orientations to key-points and then obtaining features for each key-point. SIFT uses a Difference of Gaussians (DoG) to find the key-points in images. DoG is obtained by computing the difference of a Gaussian blur of an image with two different scaling factors. Once the DoG are found, the images are searched for local extrema, which represent the set of potential key-points. This set is then refined by eliminating any low-contrast and edge key-points. SIFT then assigns the remaining key-points an orientation and a vector of features (also called descriptors) that are invariant to rotations, scale, and illumination. At the end of this process, SIFT provides the key-points and their feature vectors that will be used to compare an image to another.

Starting from the original set of frames, we use SIFT to compare neighboring frames and remove the redundant or duplicate ones, leaving a set of frames that contain unique information. In order to determine which frames to keep, we first extract the key-points from each frame and obtain their feature vectors using SIFT. Then, given two neighboring frames f_1 and f_2, for each key-point $k_{f_1,i}$ in frame f_1, we find the best matching key-point $k_{f_2,j}$ and the second-best matching key-point $k_{f_2,l}$ in f_2 based on the Euclidean distance between their features. If the Euclidean distance between $k_{f_1,i}$ and $k_{f_2,j}$ is smaller than 75% of the distance between $k_{f_1,i}$ and the $k_{f_2,l}$ (i.e., the best matching point is significantly closer than the second-best match) then the pair of key-points ($k_{f_1,i}$,$k_{f_2,j}$) is considered a strong match and added to the set $s_{1,2}$ of matching key-point pairs for f_1 and f_2. This process is called a "ratio test" and was introduced by Lowe [12]. The threshold of 75% for the distance was determined empirically, based on testing different values. This process is repeated for all key-points in f_1 and at the end the number of matched key-points in $s_{1,2}$ represents the

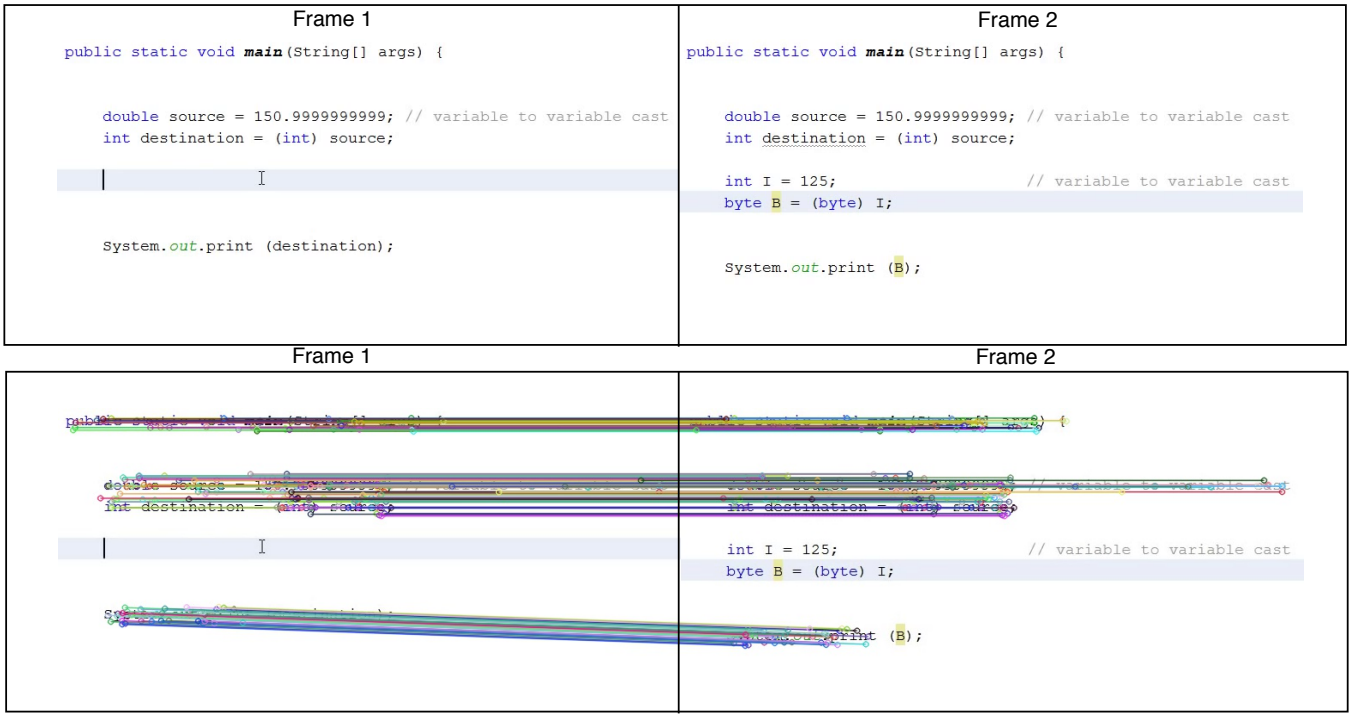


Figure 1: An example of two images matched based on key-points using SIFT

similarity measure between frames f_1 and f_2 (i.e., the more similar key-points in $s_{1,2}$, the more similar the two frames are).

Figure 1 shows a pair of similar frames extracted from a programming video tutorial. A line is drawn between each pair of matched key-points. Because there were two lines of code added in the second frame, these lines were not matched with any line in the first frame as they do not have common features. Furthermore, as the parameter of the *system.out.print* function call was changed from *destination* to *b*, there was no line drawn between the features representing the parameters.

In order to determine which frames to keep for a video, we employed the following procedure. Consider a video and its frames V = $\{f_1, f_2, \ldots, f_n\}$, and $s_{i,j}$ as the similarity value between a pair of frames f_i and f_j. We start from f_1 and compare it with its successive frame f_2 and save their similarity $s_{1,2}$. We then compare f_1 with f_3. If their similarity $s_{1,3}$ is within 10% of $s_{1,2}$ (threshold determined empirically), we consider frames f_1, f_2, and f_3 to be similar enough and continue by comparing f_1 to f_4. The comparison of consecutive frames continues until a similarity $s_{1,j}$ between the two frames f_1 and f_j differs by more than 10% compared to the first similarity value (i.e., $s_{1,2}$). In this case, the frame f_j is considered to be dissimilar to the previous ones and the following process is employed. First, frame f_1 is kept, then all the frames between f_1 and f_j are removed as they are considered similar to f_1 and each other. Frame f_j is also kept and the comparison of successive frames restarts at f_j. At the end of this process for the entire video, what is left are frames that contain considerably different information. These are the set of frames we use in the next step of our approach.

3.2 Detecting the Code Bounding Box

Our main goal in this paper is to determine the *exact location of a code snippet* in video frames, in the form of a *bounding box*. A *code bounding box* in a frame is defined as (*x, y, w, h*), where (*x, y*) is the center of the quadrilateral or box where the code is located and *w, h* are the width and height of the box. The aim is to include the entire visible source code in this box while limiting the noise (i.e., other text that is not code).

To reach our goal, we start from the set of unique frames remaining after the redundant frame removal step and then: (1) identify the frames that contain visible code, (2) locate the bounding box of the visible code. In order to determine the code bounding box, CNNs are particularly useful, as each video is a sequence of changing frames. In our case, the object of interest is the visible code bounding box. A CNN convolves throughout the entire video frame and extracts spatial features which aid towards detecting the bounding box of the code.

With CNNs, bounding box extraction could be done in two ways. One way would be to perform *pixel-wise classification* and then segment out the bounding box in the input frame using existing deep architectures like SegNet [1]. To test this approach, we performed a preliminary experiment and used a SegNet implementation[3] to train a model on a dataset of 800 frames. On testing the model on 200 unseen frames, we found the segmentation to be noisy and irregular.

A second approach would be to directly learn to predict the code bounding box without having to go through a pixel-by-pixel

[3]https://github.com/alexgkendall/caffe-segnet

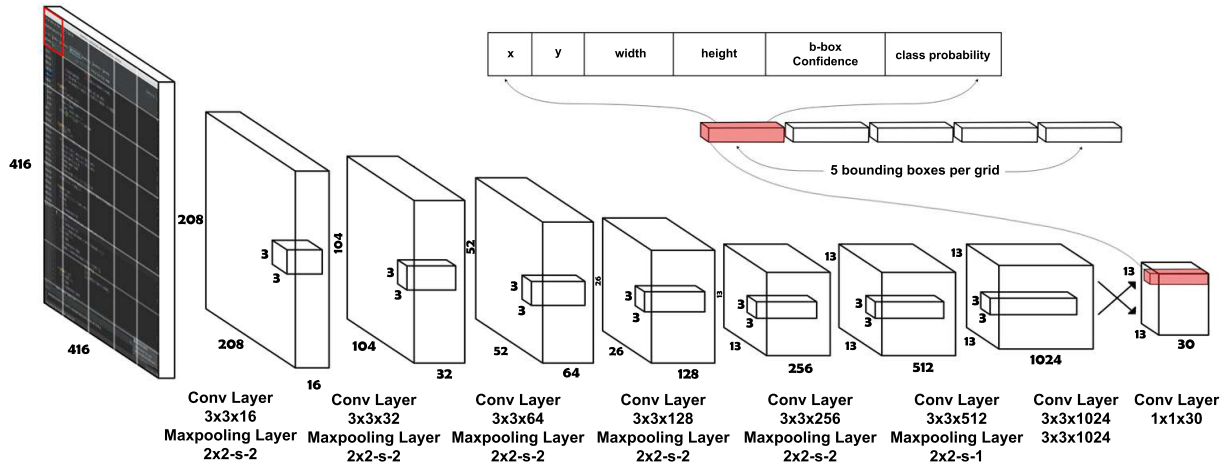


Figure 2: Our YOLO network with nine convolutional layers and six max pooling layers. The input frames are $416 \times 416 \times 3$. The output is a $13 \times 13 \times 30$ tensor (a 3-dimensional matrix) containing predictions for five bounding boxes per grid. Each bounding box prediction contains the *x* and *y* coordinates, *width*, *height*, *bounding box confidence*, and *class probability*.

classification first. In this paper, we leverage this second approach and make use of a state-of-the-art deep architecture called the **You Only Look Once (YOLO)** neural network [21] to predict the bounding box of a code section. YOLO is a unified CNN architecture which not only detects the position of the bounding box of each object present in an image but also predicts its class. Thus, YOLO looks at an input image just once and performs both detection and classification of multiple objects in one shot. Although YOLO can detect objects of multiple classes, in our case we only consider a single class: the code region.

YOLO is extremely fast since it does not require a complex pipeline and can make predictions at 150 frames per second, which makes it highly applicable to real-time video streams [21]. It analyzes the entire frame during training and test time and encodes contextual information about classes as well as their appearance. YOLO also learns the generalizable representations of objects, so it is less likely to crash when given unexpected inputs.

The YOLO Architecture:

We use the YOLO architecture given in Figure 2. We consider this shallower version of YOLO for speed and because we have only one class of objects to detect, namely the code region. The input layer accepts $416 \times 416 \times 3$ sized RGB images. The architecture has 9 convolutional layers followed by two fully connected layers. All convolutional layers have ReLU activations except for the last, which has a linear activation. As it is almost entirely composed of convolutional layers, this YOLO architecture has significantly fewer number of model parameters as compared to a neural network with fully connected layers. Each of the first five convolutional layers are followed by a 2×2 max pool layer with a stride of 2. Every such max pooling layer downsamples the input by half. Due to this downsampling, YOLO is able to utilize the global context of the entire frame to detect the code bounding box. As the frame shrinks, the filters in the deeper convolutional layers gain a wider coverage. The sixth 2×2 max pool layer does not downsample because its stride is 1. In the final fully connected layer, the input frame is divided into $S \times S$ cells. In our case $S = 13$ since we start out with an input of size 416 and halve it 5 times. For each cell, the network gives 30 predictions. Thus, the final output is a $13 \times 13 \times 30$ tensor (i.e., a 3-dimensional matrix).

The bounding box predictions and their confidences are encoded within the $13 \times 13 \times 30$ output tensor (see Figure 2). The network predicts five bounding boxes for each cell. Each bounding box prediction has six numbers associated with it: x, y, w, h, $bbox_{conf}$ and $Pr(codebox|O)$. x and y represent the coordinates of the center of the bounding box, while w and h are the width and the height of the box. If O is a candidate object that might be a code bounding box, $bbox_{conf}$ is a score encompassing two measures about this predicted bounding box: (1) its confidence of containing O, $Pr(O)$ (2) its accuracy in terms of Intersection over Union (IoU) compared to the ground truth bounding box, IoU_{gt}^{pred}.

Intersection over Union (IoU), also known as the *Jaccard index*, is a metric used to compare the similarity and diversity of sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. In our case, we divide the area of overlap between the bounding box in the prediction and the one in the ground truth by the area of the union of the bounding box in the prediction and that in the ground truth. In the frame analysis literature, when IoU is 50%, it is considered as a correct bounding box [23, 25]. However, in our case, it is really important to accurately identify the bounding box of the source code in order to ensure the extraction of correct code. Therefore, we considered a stricter rule, with an IoU greater than 75% to be a successful prediction, and anything below it to be an incorrect prediction.

$Bbox_{conf}$ is then defined by the following equation:

$$bbox_{conf} = Pr(O) * IoU_{gt}^{pred} \quad (1)$$

$Pr(codebox|O)$ measures the class conditional probability of O being a code bounding box. From all the predicted bounding boxes, YOLO picks the optimal bounding box based on the code bounding box specific confidence score given by Eq. 2.

$$\begin{aligned} \text{Confidence} &= Pr(codebox|O) * bbox_{conf} \\ &= Pr(codebox|O) * Pr(O) * IoU_{gt}^{pred} \\ &= Pr(codebox) * IoU_{gt}^{pred} \end{aligned} \quad (2)$$

YOLO optimizes a loss function composed of different types of loss: (1) SoftMax loss for the class probabilities, (2) IoU loss for the bounding box coverage, (3) mean squared loss for the bounding box centers, and (4) loss metric that puts more emphasis on the errors in smaller objects.

Even after determining the YOLO architecture, there are multiple parameters that govern how the training proceeds. We define a few of these parameters here:

- An **Epoch** is one forward and backward pass of every training sample through the neural network. Since the per batch average loss stopped oscillating after 100 epochs in our experiments, we trained our final model for 100 epochs.
- **Batch size** is the total number of training samples that will be passed through the network at once. We use a batch size of 16 (default for YOLO).
- **Subdivisions** is the amount of sub-batches that each batch will be divided into. Each can hold 8 frames.
- **Solver** denotes the gradient descent optimization method which minimizes the loss of the network that is being trained. We used RMSprop in our implementation (default for YOLO).
- **Learning Rate** is a parameter that controls how much we adjust the weights of our network with respect to the loss. A learning rate of .001 was used in our implementation (default for YOLO).

4 EVALUATION

We performed an empirical study in order to evaluate our approach on identifying the location of source code on the screen, in particular code bounding boxes in Java video programming tutorials. Our main research question is:

RQ. How well can our approach identify code bounding boxes in Java programming video tutorials?

The following subsections present our data collection, methodology, and results.

4.1 Data Collection

We first collected a set of 150 Java programming tutorials from YouTube having a high quality resolution (i.e., 1280 X 720 pixels). We manually selected these tutorials such that they contain a wide variety of topics and are created by a diverse set of tutors. We also considered videos where code was written in a variety of IDEs such as Netbeans, IntelliJ, Eclipse, etc. as well as a variety of editors such as vim, Notepad++, etc. Moreover, we included different background colors for each IDE and code editor (i.e., white and black). It was important to have a mix of configurations between our videos in order to make our model more robust and accurate. Another aspect we looked at when it came to IDEs was their layout. IDEs usually have at least 3 different sections: the main editing window, the file explorer and the output window. We made sure to find videos using differently sized and shaped sections in the layout. We also included videos that had different numbers of sections in the layout.

We downloaded the videos using the Python library PyTube[4]. The length of the videos we collected this way varies between 12 and 3,327 seconds, with an average of 548 seconds. We make our dataset, scripts, and results available in our *replication package*[5] (17GB in total).

Note that although the collected dataset for this study contains only Java programming video tutorials, *our code bounding box detection approach can be applied to videos created for any programming language* by training the model with video tutorial examples of that programming language.

4.2 Methodology

We extracted one frame per second for the videos we collected and obtained 82,335 frames in total from all 150 videos. We then applied the redundant frame removal algorithm described in Section 3.1, which resulted in over 70% of the frames being removed. The number of frames were therefore reduced to only 23,576 as shown in Table 1. On average, we reduced the number of frames in the original set from 548 to 157 per video, leading to a much reduced analysis time.

Table 1: Results of the redundant frame removal step (fpv=frame per video)

	Total frames	Max fpv	Min fpv	Average fpv
Original	82,335	3,327	12	548
Selected	23,576	927	5	157
Deleted	58,759	2,400	7	391

After removing redundant frames, we asked developers to classify each remaining frame according to the visibility of the code in it. More specifically, we considered three classes:

- **Fully-Visible Code (*FVC*).** A frame contains fully visible code if the main editing window is not obstructed in any way and every line of code is clear and readable.
- **Partially-Visible Code (*PVC*).** A frame contains partially visible code if the main editing window is visible, but it is obstructed in some way (drop down menu, settings pop-up, etc.) or the code is obstructed by something like code completion suggestions.
- **No Code (*NC*).** A frame is considered to contain no code if it does not contain a main code editing window. These could be frames from a video where the tutor is explaining a topic on slides or a board, is searching in their browser, or intro and exit scenes, etc.

To make the frame classification process easier for the participants, we created a web-based classifier. This tool allowed multiple participants to classify frames in parallel. We had a total of 10 participants classifying the frames. All were Computer Science students, with six being PhD students, two MS students, and two BS students. Each participant was presented with one frame at a time and asked

[4]https://github.com/nficano/pytube
[5]https://goo.gl/R7yGiD

Table 2: Manual Classification Results (FVC=Fully-Visible Code; PVC=Partially-Visible Code; NC=No Code)

		First Participant		
		FVC	PVC	NC
Second Participant	FVC	14,081	78	27
	PVC	150	1,565	144
	NC	69	103	7,359

to classify it with the class they believed most accurately described the frame (FVC, PVC, or NC). The web classifier utilized SQLite to maintain a database of each participant's classifications.

To ensure accuracy, we had two participants independently classify each frame, so each participant was assigned a fifth of the 23K remaining frames. Table 2 shows the results of the manual classification, specifically where the pairs of participants agreed and disagreed about the classification of a frame. In particular, the diagonal of the table shows the number of frames where the pair of participants assigned to a frame agreed on its classification. The numbers outside the diagonal represent the cases where the two participants had a disagreement about the classification of a frame. The least disagreements occurred when one participant classified a frame as having fully-visible code (FVC) and the other as having no code (NC). This is to be expected as these two classes of frames are very different and easily distinguishable from each other.

We then discarded any frames that had different classifications between a pair of participants. After discarding these cases we were left with 14,081 frames classified as having *fully-visible code*. We did not consider frames having only partially visible code for the rest our study, as our main focus is identifying bounding boxes that contain fully visible code that can be eventually extracted for reuse.

Ensuring Balance and Variety in the Training Set:
To train a robust model, we should ensure that the frames in the training set are balanced in terms of the occurrence frequency of their class. In addition, having a variety of frames within every class is also advantageous since the model can learn from a more diverse set of examples.

Since annotating the entire set of 14,081 frames having fully-visible code with their respective code bounding boxes was unfeasible, we aimed at sampling 2,000 frames to annotate and later train our deep learning model on. We also aimed at selecting 2,000 frames from the set classified as not containing code, in order to have counter-examples to train our deep model with. However, in order to ensure diversity in the training set, we did not select these frames completely randomly from the two sets, but first applied a simple clustering technique to the 14,081 frames. Clustering was used to group the similar frames together and allowed us to maximize the diversity of our training set by ensuring we choose frames from all the different clusters.

We clustered the frames in a lower dimensional space because their original size was too large to efficiently process. We reduced the frame dimensions in two stages: (a) we resized all frames to 416× 416, which is also the input size expected by YOLO (see Figure 2) (b) we then projected the 416×416 frames into a low pc dimensional space. For the latter, we first selected a small, random sample of 1400 frames from the set of 14,081 resized frames containing fully visible code (almost 10%). We then performed Principal Component Analysis (PCA) on these frames and sorted and visualized the top 100 Eigen values. Based on these results we chose $pc = 12$ as the number of principal components, such that the majority of the variance is preserved. We then projected all 14,081 frames to a 12 dimensional space using the first $pc = 12$ principal components. We then applied the KMeans algorithm [7], a popular clustering scheme, with $k = 1,000$ to cluster the frames in this reduced space and obtained 1,000 clusters. After clustering, we randomly selected 2 frames from each cluster. This resulted in a diverse set of 2,000 frames with fully-visible code that were ready to be annotated with code bounding boxes. We similarly also clustered the 7,359 frames containing no code into 1,000 clusters and randomly selected 2 from each cluster to get our set of 2,000 no-code frames serving as counter-examples. We can see in Table 3 the min, max, and average frames per cluster between the 1,000 clusters in each of the two categories (FVC and NC).

Table 3: Statistics for the 1,000 clusters in each category FVC=Fully-Visible Code; NC=No Code; fpc=frames per cluster

	Total frames	Max fpc	Min fpc	Average fpc
FVC	14,081	149	1	14
NC	7,359	108	1	7

Annotating the FVC frames with Bounding Boxes:
To make annotating the frames with bounding boxes easy for developers, we implemented an annotation tool. This tool allowed the user to click and drag a box over the code section of each frame. The box could then be edited or accepted. Once accepted, the tool would save the relevant data to an XML file. This XML file contained important information about the annotation of a frame such as the directory in which the frame resided, the filename of the frame, the width and height of the frame and the code bounding box of that frame. The bounding box is defined by ($xmin$, $ymin$, $xmax$ and $ymax$). These four values represent the top left and bottom right corners of the bounding box. The 2,000 non code frames needed to be annotated as well, but their annotations are all the same since there is no code bounding box at all. To annotate these we just set all x and y min and max values to 0.

To annotate all of the fully-visible code frames, we recruited the same group of 10 students that classified the initial set of frames to take on this task. We divided the set of frames into 5 subsets, where each student was assigned a subset to annotate. We made sure that each subset was annotated separately by two different participants so that we could check for accuracy. Each student then used our annotation tool to annotate their set, and submitted their annotations to us.

To ensure we had accurate annotations, we had two different students annotate a set of frames and then validated that both of the annotations were in agreement on where the bounding box should be. To validate that both students agreed on the location of the bounding box we created a script to compute the IoU between both students' annotations. If the IoU of a frame had a value below 90% then the frame was discarded, as the students did not fully agree on the location of the bounding box. If the IoU value was 90% or above, we randomly selected one of the annotations, as they were almost overlapped.

Metric:
We used the Intersection over Union (IoU) metric [25] to validate our model and measure the accuracy of a predicted code bounding box. Formally, IoU divides the area of overlap between the bounding boxes of the prediction and ground truth by the area of the union of the bounding boxes of the prediction and ground truth. When the IoU is 50%, it is considered as a correct bounding box [10, 25]. However, in our case, it is very important to accurately identify the bounding box of the source code in order to ensure a correct code extraction and reuse later on. Therefore, we considered a prediction with an IoU of greater than 75% to be a successful prediction, and anything below it to be an incorrect prediction.

Let A_{gt} and A_{pred} be the area of the ground truth and the predicted bounding boxes respectively. IoU is given by the equation:

$$IoU = \frac{A_{gt} \cap A_{pred}}{A_{gt} + A_{pred} - (A_{gt} \cap A_{pred})} \tag{3}$$

$A_{gt} \cap A_{pred}$ represents the area of the overlap between the predicted and the ground truth bounding boxes. In Equation 3, IoU is computed by dividing the area of intersection by the area of the union. This results in a ratio between 0 and 1, where 1 denotes a perfect overlap. In other words, if the ground truth bounding box has exactly the same coordinates as the predicted bounding box, then $IoU = 1$.

We have to adjust the IoU computation to accommodate the cases when the ground truth of the test frame does not have any annotated code bounding box in it. Otherwise, Equation 3 would give a "division by zero" error. The IoU adjustments are as follows:

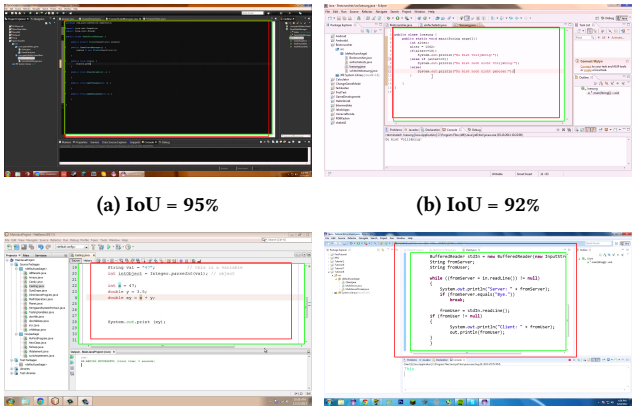
- If $A_{pred} = 0$ and $A_{gt} = 0$, then $IoU = 1$. This happens when our model predicted that there was no bounding box and it was correct.
- If $A_{pred} = 0$ and $A_{gt} \neq 0$, then $IoU = 0$. This happens when our model did not predict any bounding box but the ground truth actually had one.
- If $A_{pred} \neq 0$ and $A_{gt} = 0$, then $IoU = 0$. This happens when our model predicted there was a bounding box when there actually was none. So, it predicted incorrectly.
- If $A_{pred} \neq 0$ and $A_{gt} \neq 0$, then IoU is calculated according to *Equation 3.*

4.3 Results

We conducted all the experiments on a machine with an Intel Xeon 3.40GHz processor, 128GB RAM, and a GeForce GTX 1080 GPU with 8 GB of memory.

Training: We used the dataset we created previously in our evaluation, which has two sets (fully visible code and no code) of 2,000 frames each and their annotations. We split each of the two sets separately at random and kept 80% for training and used the rest of 20% for testing. We then merged the two training sets and separately the two test sets such that each includes both FVC and NC frames. Finally, we obtained training and testing set sizes of 3,200 and 800 respectively.

Testing: We applied the trained model on the test dataset that includes 800 frames. For each test frame, we computed the IoU as described in Section 4.2. The results indicated that our approach



(a) IoU = 95% **(b) IoU = 92%**

(c) IoU = 82% **(d) IoU = 73%**

Figure 3: Predicted code bounding boxes of three sample frames from different videos

was extremely effective at detecting the exact code bounding box, having on average the IoU of the testing set **92%**.

We present a few example predictions in Figure 3. Each frame shows their respective ground truth and predicted bounding boxes. The green bounding box represents the ground truth while the red bounding box represents the prediction. Figure 3a has an IoU of 95%; the predicted bounding box is almost exactly the same as the ground truth bounding box. This may be because the main editing window is a different shade of gray, so it is very clear there is a box and the model was able to find the window very easily. On average, we have a prediction accuracy that looks like Figure 3b which has an IoU of 92%. Again, the predicted bounding box is very close to the ground truth box. This frame, like Figure 3a and Figure 3c, has a good amount of contrast between the code editing window and the surrounding features. These are ideal predictions to eventually use OCR on. We can see in Figure 3d that the red and green bounding boxes do not match up quite as nicely as we would hope. The IoU for this frame is 73%. We consider this a poor prediction, as the bounding box covers more information than we would want. If we were to use OCR on this bounding box we would pick up text from the file tabs that contain filenames. This may be predicted poorly because there is not much contrast around the area of the box, i.e., it is hard to locate the box.

5 LIMITATIONS OF OUR APPROACH AND POTENTIAL IMPROVEMENTS

Although the network successfully predicts the bounding box of code, there are some limitations as follows. First. there are still a few cases in which the approach predicts the bounding box for an image poorly. In this paper we considered an IoU of below 75% to be a poor prediction. One potential reason for a poor prediction could be that the network was not trained enough on a particular type of image. This could be probably addressed by including more images similar to the bad predictions in the training set. We believe doing so would result in better predictions overall. Another way to solve this problem could be through the use of contours. A contour can be explained simply as a curve joining all the continuous points along

a boundary that have the same color or intensity. For example, if an image has a confidence value (described in Section 3.2) of below 30%, then we could consider this method. Once we have an image with a low confidence value like this, we could check to see if there is at least one contour for it. If so, we could compute the IoU of each contour with the predicted bounding box and consider the highest IoU that is above 75% to be the prediction. This threshold is to ensure that we would not consider invalid contours, or ones that do not cover the editing window.

While it has proven relatively easy for people and machines to distinguish between a frame containing source code and a frame with no source code at all, this is much more challenging when comparing frames containing fully-visible source code to those with partially-visible source code. Fully-visible code could be truncated by only few words or even a mouse cursor. In these cases, we had questions from participants if they should consider these frames as containing fully-visible or partially-visible code. When it comes to our approach, it failed to distinguish between these types of frames in some cases, such as when having a mouse cursor hovering over the code. Yet, in other cases, when the approach predicted a bounding box for a frame that has code obstructed by a drop-down menu, a new opened window, etc. it successfully returns an area of zero that indicates the obstruction of the code bounding box.

For now, our approach can predict the bounding box successfully for the frames that contain typed code. However, we have not tested it on predicting the bounding box of handwritten code. Handwritten code can be on a board, piece of paper, etc. We did not yet consider handwritten code in our dataset, but we plan to analyze such frames in the future.

In our dataset, we have a few IDE frames with no source code written in them. These were classified by the participants as having no code. However, as the overall features of these frames look very similar to the ones of frames containing code, the model predicts a code bounding box. This, however, may not be a major issue, since the OCR will not extract anything from this kind of box, as it contains no text.

6 THREATS TO VALIDITY

The threats to *internal validity* in our study include the impact of learning and fatigue on the quality of the participants' responses. To mitigate this, we had two people classify each image and define each code bounding box and only used those images and bounding boxes on which the participants agreed. In order to mitigate the potential learning effect, we made sure the different frames seen by each participant varied across videos.

Regarding the threats to *external validity*, our results may not be generalizable to all the software development videos available or all their frames. However, we aimed at increasing the generalization of the findings for Java tutorials by considering 150 different videos on a wide variety of topics and created by numerous developers. This is the largest set of videos considered so far in studies concerning the identification or extraction of source code from video tutorials. In our evaluation we also made sure we select a diverse set of frames, covering all 150 videos.

Threats to *construct validity* refer in our case to how we measured the effectiveness of the approach. We mitigated threats to construct validity by employing the IoU metric to measure the overlap between the predicted code bounding box and the ground truth, which is a well established measurement in the field of object identification in images [10, 25].

7 RELATED WORK

Software development video tutorials are becoming a form of documentation that developers create and also consult to support their programming tasks [13, 18]. Studying the motivation that developers have to create these forms of documentation, MacLeod et al. [13] found through a set of interviews that one of the goals is sharing knowledge they gained while performing a programming task. Other work analyzing video tutorials focused on tagging video tutorials with expressive labels that could help developers decide if a tutorial is helpful or not for their task at hand [4, 16]. Poché et al. [17] leveraged the term frequency of words in the comments of software development videos to identify and extract general user concerns about the video. Ellmann et al. [3] studied the properties of programming screencasts in order to understand how similar or different they are from other types of screencasts, and found that programming tutorials are more static than others (i.e., the images change at a slower rate). This is an important finding for research analyzing programming video tutorials, as it indicates that many frames in such screencasts are duplicates. This also motivates the frame duplicate removal step in our approach.

The most related to our approach is, however, work that identified or extracted code from programming video tutorials [15, 19, 20, 24]. Ponzanelli et al. [19, 20] introduced CodeTube, web-based tool that enables developers to query software development videos and view only the specific fragments related to their query. As part of indexing the video tutorials' content for search, the authors also extract any Java source code that they identify on the screen. However, to identify the part of the screen that contains the code, the authors use heuristics and make assumptions that do not always hold and can lead to incorrectly extracted code. One example is the heuristic which divides the image into fixed-size sub-images and then selects the one having the most Java keywords as containing the code. As discussed in their paper, this can lead to incorrectly identifying an entire IDE window or a part of it such as the package explorer as containing the source code, or truncating the code that did not fit into the fixed-size image. This, however, is not a deal-breaker in the case of CodeTube, which uses the extracted text and code only as supplemental information for indexing a video, in addition to the video's description, transcript, etc. In our case, however, things are different. We aim to determine the location of the code as accurately as possible, since our goal is to enable the correct extraction of source code to support its reuse. Our deep learning approach does not use any kind of assumptions on the size or location of the code on the screen and is therefore able to overcome the limitations that CodeTube faces.

One other closely related work is that by Yadid and Yahav [24], which introduce ACE, a tool that combines language models and image processing techniques to extract source code from software development videos. In order to extract the code from a video, they initially identify the location of the main editing window in only one frame of the video and then set that location for all other frames in the video. While this is convenient and computationally efficient, it can also be incorrect in many situations, when the window is moved,

split, resized, overlapped, etc. This can be problematic when the goal is to extract correct code throughout the video. Our approach overcomes these limitations by extracting the location of the code for each frame independently, and does not rely on assumptions about its location.

Finally, in a recent work by Ott et al. [15] the authors applied a CNN to classify the presence or absence of code in about 20K frames extracted from 40 programming videos. While they also use deep learning as their main approach, our work is different than theirs in several ways. The most important difference is that Ott et al. do not identify the location of the code in video frames, but rather just its presence or absence. We go one step further and identify the exact location of source code within each frame, which is an essential step towards the extraction of reusable source code from videos. One other important difference is that we employ a duplicate detection and removal mechanism in our approach, which significantly reduces the number of frames to be analyzed within a video. This allowed us to increase the number of videos used in our study to 150, ensuring more diversity in the dataset.

8 CONCLUSIONS AND FUTURE WORK

In this paper we proposed a novel approach for detecting the precise location of source code in the video frames of programming tutorials. Our approach first identifies and removes duplicate frames from a video and then uses a Convolutional Neural Network to identify the precise bounding box where source code is being displayed on the screen. We performed an evaluation study on a set of 4,000 frames extracted from 150 Java programming videos. The frames were manually annotated by developers with the correct code bounding box and used as ground truth in the evaluation. When comparing the predicted code bounding box and the ground truth, our approach was able to correctly predict the precise location of the code with a 92% accuracy. To the best of our knowledge, this is the first approach that is able to detect the location of source code with such accuracy and without relying on heuristics or assumptions about its location.

Our long-term goal is the extraction of correct, reusable code from software development video tutorials, in order to allow developers to easily integrate this code into their programs. Our future work will move forward in this direction, focusing on extracting the source code from the area determined by our approach, and then correcting it and augmenting it as needed, until the end result can be efficiently reused.

ACKNOWLEDGMENTS

Sonia Haiduc is supported in part by the National Science Foundation grant CCF-1526929. Mohammad Alahmadi is sponsored in part by the Saudi Arabian Cultural Mission (SACM) and University of Jeddah (UJ).

REFERENCES

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. Segnet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 39, 12 (2017), 2481–2495.

[2] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.

[3] Mathias Ellmann, Alexander Oeser, Davide Fucci, and Walid Maalej. 2017. Find, Understand, and Extend Development Screencasts on YouTube. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*. ACM, 1–7.

[4] Javier Escobar-Avila, Esteban Parra, and Sonia Haiduc. 2017. Text Retrieval-based Tagging of Software Engineering Video Tutorials. In *Proceedings of the International Conference on Software Engineering*. 341–343.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[6] Adam Grzywaczewski and Rahat Iqbal. 2012. Task-Specific Information Retrieval Systems for Software Engineers. *J. Comput. System Sci.* 78, 4 (2012), 1204–1218.

[7] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108.

[8] Wei Hu, Yangyu Huang, Li Wei, Fan Zhang, and Hengchao Li. 2015. Deep convolutional neural networks for hyperspectral image classification. *Journal of Sensors* 2015 (2015).

[9] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. 1999. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*. Springer, 319–345.

[10] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. *ArXiv Preprint arXiv:1708.02002* (2017).

[11] David G Lowe. 1999. Object Recognition from Local Scale-Invariant Features. In *Proc. of the 7th IEEE Int. Conf. on Computer Vision*, Vol. 2. IEEE, 1150–1157.

[12] David G Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.

[13] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, Camera, Action: How Software Developers Document and Share Program Knowledge Using YouTube. In *Proc. of the Intl. Conf. on Program Comprehension*. 104–114.

[14] 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.

[15] Jordan Ott, Abigail Atchison, Paul Harnack, and Erik Bergh, Adrienne adn Linstead. 2018. A Deep Learning Approach to Identifying Source Code in Images and Video. In *IEEE/ACM 15th Working Conference on Mining Software Repositories*. 376–386.

[16] Esteban Parra, Javier Escobar-Avila, and Sonia Haiduc. 2018. Automatic Tagging for Software Engineering Videos. In *Proceedings of the International Conference on Program Comprehension*. 222–232.

[17] Elizabeth Heidi Poché. 2017. *Analyzing User Comments On YouTube Coding Tutorial Videos.* mathesis. Louisiana State University, Baton Rouge, LA, USA.

[18] 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 *Proc. of the Intl. Conf. on Software Engineering*. ACM, 261–272.

[19] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Codetube: Extracting Relevant Fragments from Software Development Video Tutorials. In *IEEE/ACM International Conf. on Software Engineering Companion*. IEEE, 645–648.

[20] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, Sonia Cristina Haiduc, Barbara Russo, and Michele Lanza. 2017. Automatic Identification and Classification of Software Development Video Tutorial Fragments. *IEEE Trans. on Software Engineering* (2017).

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning Internal Representations by Error Propagation.* Technical Report. California Univ San Diego La Jolla Institute for Cognitive Science.

[23] Ray Smith. 2007. An Overview of the Tesseract OCR Engine. In *Ninth International Conference on Document Analysis and Recognition*, Vol. 2. IEEE, 629–633.

[24] 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.

[25] C. Lawrence Zitnick and Piotr Dollár. 2014. Edge Boxes: Locating Object Proposals from Edges. In *European Conf. on Computer Vision*. 391–405.