Deep Learning on the Web: State-of-the-art Object Detection using Web-based Client-side Frameworks

Xenofon Pournaras
Computer Engineering and Informatics Department
University of Patras
Patras, Greece
pournaras@ceid.upatras.gr

Dimitrios A. Koutsomitropoulos
Computer Engineering and Informatics Department
University of Patras
Patras, Greece
koutsomi@ceid.upatras.gr

Abstract

In the present paper we make a comparative study and evaluation of frameworks and libraries for deep learning purposes on the client-side, considering libraries such as TensorFlow.js, brain.js, Keras.js, ConvNet.js and others. It is examined how feasible and efficient it is to execute deep learning tasks, using client-side libraries and frameworks in contrast to the conventional approach. Moreover, we focus on the computer vision field of object detection and we examine the problem of object detection through different state-of-the-art approaches and object detectors. At the same time, we evaluate whether it is feasible and efficient to detect objects in the browser environment using a prototype implementation based on some of the libraries that are studied.

Keywords – Object Detection, In-browser Deep Learning, Deep Learning Frameworks, Machine Learning

I. Introduction

The resurgence of artificial neural networks and the combination of deep learning techniques in recent years seem to have promising results in approaching traditionally intractable problems. The algorithms proposed for the development and training of such models require a large computational power found either in supercomputers or in distributed systems optimized for this purpose. As a result, there is no expected access to such tools by those interested inside the scientific community, who want to develop and train models of deep learning.

However, very recently, deep learning applications aiming at everyday computer use, have begun to emerge, specifically in Internet browsing, to give a sense of intelligence to the user's browsing experience. More specifically, the emergence of libraries that support a wide range of functions for various deep learning tasks in the browser has managed to create a favorable climate for the creation of such applications with the use of conventional computing resources. Recent developments in browser technology have contributed greatly to several improvements in the performance of models and their ability to implement, train and run them. Most modern browsers support WebGL [18] and WebGPU [4] technologies, with which they can access and use the graphics card to speed up the operations of tensors and matrices. Deep learning libraries such as TensorFlow.js [24] and WebDNN [16] support and use some of the above technologies. Applications and models that use few computing resources can be trained and run on the browser and not on a powerful computing system or an expensive graphics card.

Furthermore, the user does not have to install local libraries or library dependencies to receive packages for each program to work. In this way the portability aspect is increased. Given the fact that deep learning algorithms or applications are implemented on a web environment, this means that they can be accessed and run easily and quickly by users. Additionally, all data remain on the user's side. In this way, privacy is protected. Taking into consideration that browsers offer access to an interactive space including cameras and microphones, developers can easily take advantage of the resources of each device to create deep learning models that serve different purposes.

Recent frameworks supporting multiple types of artificial neural networks, especially Convolutional Neural Networks (CNNs), make the delivery and execution of object detectors in the browser environment possible. Frameworks such as TensorFlow.js [24] and others can help developers create their own custom object detectors and run them on the Web.

In this paper we first examine the object detection problem through various well-known object detectors. Then we review and survey various web-based JavaScript frameworks that can be used to implement and execute deep learning tasks on the client side. Based on these, we propose an implementation of a web application[1] which can detect objects in the browser with the use of some deep learning models that we have described. This implementation has been also customized and tested on mobile, handheld devices (smartphones) as a web app. Finally, we present some results related to the performance of the object detectors in the browser based on the implementation that we have created.

To the best of our knowledge, a qualitative survey of web frameworks focusing on deep learning object detection has not been investigated before. In addition, the implementation and experimental evaluation solely on the client-side of both object detection models, namely Tiny-YOLO [21] and COCO-SSD [24], supporting multiple features including snapshot object detection, video detection and detection of imported images, has not appeared previously.

The rest of this paper is organized as follows: In Section II we examine well-known object detectors. In Section III we present our comparative survey among web-based frameworks that are used for deep learning purposes. Next, in Section IV we describe the prototype application for in-browser object detection. Section V contains our experiments and results for evaluating in-browser object detection based on our implementation. Finally, Section VI summarizes our conclusions and future work.

[1] http://github.com/xenofonpournaras/in-browser_object_detection

II. Deep Learning Object Detectors

With the appearance of a remarkable number of frameworks and libraries in the JavaScript environment for deep learning purposes, object detection can occur effectively. To investigate object detection in the browser we must divide it into two aspects and analyze each of them separately. The first aspect includes state-of-the-art object detectors, which are based on CNNs and are widely used to efficiently address the problem of object detection. The second aspect is the proper functionality and the efficiency of these models running in the browser.

A. Object Detectors

Object detectors can be divided into two categories based on the number of stages that perform the predictions namely, two-shot detectors and single-shot detectors. Two-shot detectors perform the detection in two stages such as region proposal and then the classification of those regions. The other category is the single-shot detectors, which perform the localization and the category prediction in one stage, skipping the region proposals. For a detection model to make precise and fast detections, a reasonable number of regions must be selected, taking into consideration spatial locations, different aspect ratios as well as the computational cost of those.

The first efficient two-stage method proposed was Regions with CNN features (R-CNN) [6] which uses a selective search algorithm to extract 2000 regions from the image. These region proposals are fed into a CNN so that features can be extracted from the image. The output of each CNN is then fed into a Support Vector Machine (SVM) to classify objects within the candidate region and a linear regressor is used to adjust the bounding box precisely. This procedure happens for each region. The major drawback of R-CNN is that is very slow in training as well as in the testing of new images.

The major disadvantages of R-CNN were solved by the Fast R-CNN [7]. The approach is similar, but Fast R-CNN has two significant changes in the way it operates. The first change is that instead of feeding each region of proposal to a CNN, now the input image is fed into a CNN to extract a feature map. The second change is the substitution of SVMs for a softmax layer, which is used for the prediction of the class of the region proposal. Fast R-CNN is significantly faster than R-CNN because the process of detection consists of only one CNN for the entire image and one layer for the predictions, in contrast to R-CNN which uses one neural network with many SVMs for the prediction.

The successor of Fast R-CNN was Faster R-CNN [22]. This object detector shares the same approach as the Fast R-CNN. The major improvement of Faster R-CNN is the use of a region proposal network (RPN) instead of the selective search algorithm. As a result, Faster R-CNN ends up 10x faster than Fast R-CNN. Therefore, it can be used for video detection but with a low frame rate.

B. Single-Shot Detectors

A state-of-the-art single shot detector is YOLO [20]. YOLO approaches the object detection problem as a regression problem and gets the detection in a single pass. YOLO divides the input image into a grid of S x S cells.

Each grid is responsible for predicting a fixed number of bounding boxes, if the center of those fall within that grid. Each grid also predicts a confidence score for the bounding boxes. A class probability map is also used for the classification of each box. The bounding boxes with their confidence and the class probability map are combined by the network, to make the prediction of the classes and the offset values of the bounding boxes. During the process, bounding boxes with low confidence scores are removed. The major advantage of YOLO model is that it is very fast as it can run at 45 FPS on a computer with a GPU, which makes it possible to make real-time detections with notable precision. Predictions are made from one single network and the model can generalize better. A limitation of YOLO algorithm is that it has a hard time when it comes to detect small objects due to spatial constraints.

TABLE I. Inference Time and Mean Average Precision of Single Shot Detectors in Various Datasets

MODEL	**DATASET**	**mAP**	**Inference time (ms)**
YOLO	VOC2007	63.4%	22
YOLO	VOC2012	57.9%	29
SSD300	VOC2007	74.3%	21.7
SSD512	VOC2007	76.8%	52.6
YOLOv3-416	COCO@mAP$_{0.5}$	55.3%	29
YOLOv3-320	COCO@mAP$_{0.5}$	51.5%	22
TINY YOLO	COCO	23.7%	4
TINY YOLO	VOC2007	57.1%	4.8

However, the latest version YOLOv3 [19], resolves this issue. Firstly, the architecture of the network consists of 106 layers. Secondly residual skip connections and upsampling are used. The major improvement is that the model performs predictions at three different scales as kernels are used in three different feature maps at three different positions in the network. Moreover 9 anchor boxes are grouped into 3 different groups according to their scale and are used to a specific feature map. Finally, it performs multilabel classification instead of using softmaxing classification. All the above, contribute to the detection of small objects, to a better accuracy performance and to a more robust detector. Due to its more complicated architecture, YOLOv3 achieves 20 to 45 FPS depending on the size of the input image.

TINY YOLO [21] is a lighter version of YOLO making it faster in object detection. There are several versions of Tiny YOLO. This object detector is quite efficient for object detection in the browser environment because of its small and powerful architecture. The weights file is about 40MB. TINY YOLO follows the YOLO detection algorithm and the network consists of 9 convolutional layers and 6 pooling layers. In terms of speed Tiny YOLO achieves 200 to 244 FPS on a GPU depending on which dataset it is trained. But there is a tradeoff between speed and accuracy. On COCO dataset [13] it achieves 23.7% precision.

SSD [14] is another single shot detector that predicts bounding boxes and the class in a single pass and has a great accuracy on objects with different scales. The architecture of SSD usually starts with a deep network base model which is converted to a fully convolutional neural network. More convolutional layers are then added which produce feature maps of multiple different scales for detection. For each of these feature maps a convolutional filter is used to evaluate a set of anchor boxes. The model predicts the class probability and the offset values of the bounding boxes, for each anchor box. Non-maximum suppression algorithm is used for the final detections. Any base network model can be used as a feature extractor such as VGG [1], ResNet [10] etc. The combination of a MobileNet [11] as a feature extractor with the SSD architecture is an efficient object detector for mobile vision applications. Table I shows a comparison of the inference time and the mean average precision (mAP) of the single shot detectors discussed.

C. *Object Detectors for the Web*

During the client-side use of the model, the model is embedded inside the front-end instead of being loaded and run in the back end. Thus, the model is loaded and runs inside the browser. Browser constraints, the computational power of the device that is running the model as well as the size of the model are key parameters for the efficient running of the model and the normal utilization of the user's resources. Another key parameter is the support of the structure and the processes of CNNs from libraries that are used either for the training or the inference phase of the model.

The running of large size object detectors is prohibitive for the browser environment because of the browser constraints on the GPU usage. For the efficient running of the object detectors it is necessary for them to occupy a normal space in the CPU or the GPU so that they can be exploited properly both by the browser and the resources of the device. If we are interested in speed and high FPS, it is important that the size of the model is small in order to make a relatively small number of FLOPS and it does not consist a workload for the hardware of the device.

Current libraries and frameworks, that support client-side training and inference phase of object detectors, are slower than their counterparts that are used for native applications [15]. Also, the majority of those do not support all the features needed either to import a well-known pre-trained object detector or to create a complicated object detector due to lack of some functions. However, frameworks like Tensorflow.js and WebDNN support almost all the necessary functions and processes to train and run an object detector with the use of WebGL, WebAssembly and WebGPU in an efficient way. However, due to their recent appearance they cannot achieve the same results as the frameworks used for native applications. So, it is imperative to take into consideration the architecture of the object detectors, the datasets they are trained with, the browser constraints and the workload when it comes to training and running an object detector.

III. Review of Web-Based Frameworks for Machine Learning

In this section we make a comparative survey between a wide range of frameworks which are used for machine and deep learning purposes in the browser environment. We introduce the frameworks selected for the study and we compare the features, the functionality as well as the support that they offer to the developer community.

TensorFlow.js [24] is an in-browser machine learning library released by Google. TensorFlow.js is a powerful library which supports the creation, the training as well as the running of machine learning models in the browser. It provides an API for linear algebra operations between vectors, matrices, and tensors as well as a higher-level API for model creation, training, and inference with emphasis on neural networks. Tensorflow.js is designed to run on both a browser and a server. When it runs on a browser, it takes advantage of the device's GPU via WebGL for the parallel processing of operations. It provides a wide range of supported layers, activation functions and optimizers for all kinds of deep learning networks. It also provides interconnectivity with other frameworks such as Keras and TensorFlow which makes easy for the developer to import pre-trained models for the training and the inference phase. Moreover, it provides the necessary API to load and save models for future use. Regarding developer support it provides a detailed documentation as well as several demos.

brain.js [2] is an open source JavaScript library which also supports creation, training, and running machine learning models. It supports network types such as Deep Neural Networks (DNNs), Recurrent Neural Networks (RNNs) and Long Short-Term Memory Neural Networks (LSTM). Compared to TensorFlow.js it does not offer neither CNN type support nor acceleration via WebGL. Instead it offers GPU acceleration with the GPU.js library. When the GPU is not available it offers parallel processing with the use of the CPU. It also provides saving and loading models with JSON. The major advantage of this library is that it is very simple to use. The purpose of this library is to simplify the complexity of deep learning and machine learning models. It is especially useful for developers entering the field of machine learning. The user can easily create and train a neural network, adjusting parameters such as momentum and learning rate without having deep knowledge of deep learning models. It also supports a notable number of layers, activation function and optimizer types. Regarding the developer support it provides demos as well as online tutorials.

ConvNet.js [3] is a deep learning library developed in JavaScript aiming at the development of deep learning models entirely in the browser environment. It provides support for creating, training, and running models in the browser. DNN's, CNN's, classification with SVM's, regression models as well reinforcement learning are some of the deep learning models that are supported. Several layer, optimizer and activation functions are also supported for the development of deep learning models. An API for the loading and the saving of models is also supported. This library does not offer support neither for RNN networks nor for acceleration via WebGL. However, it is a powerful and well-known library. A major disadvantage of this library is that it has not been maintained for some years. Regarding developer support it provides documentation as well as demos.

Keras.js [8] is a framework that allows import and running of pre-trained deep learning models. It is used only for the inference phase as it does not support training of neural networks. It provides acceleration via WebGL 2. Models can also be run in Node.js but only with the exploitation of the CPU. Keras.js abstracts away several frameworks as backends. In this way models can be trained in any back-end, such as TensorFlow, CNTK etc. Keras.js also provides several pre-trained models and mainly CNNs such as Inception v3 [23]. Regarding the developer support it provides only demos. A major disadvantage of Keras.js is that it is no longer active, and its demos are no longer updated.

Synaptic.js [9] is a JavaScript neural network library both for the browser and Node.js. It provides several built-in architectures such as Perceptron, multi-layer Perceptron, LSTM, LSM, Hopfield networks, self-organizing maps, and liquid state machines. It supports deep neural networks and first and second order RNNs. It does not provide support for CNNs. It is not as sophisticated as TensorFlow.js as it is slow due to lack of WebGL support and offers a limited number of layer and activation function types. However, it is a new library that is maintained and constantly evolving. Regarding the developer support it provides detailed documentation and several demos.

WebDNN [16] is an open source framework developed by the University of Tokyo. The purpose of this framework is to optimize the execution of pre-trained deep learning models in the browser. Thus, it supports inference tasks only. It also supports acceleration via backends such as WebGL, WebAssembly, pure vanilla JavaScript implementation and is one of the few frameworks that supports WebGPU. WebDNN optimizes the process of the execution of the model by compressing the model data with a unique technique, taking advantage of the described APIs. It supports all types of networks and can execute models trained from various frameworks such as Keras, TensorFlow, Chainer and Caffe. According to empirical evaluations WebDNN is significantly faster than Keras.js in terms of speed. In some cases, it achieved more than 200x acceleration when WebGPU and optimization were used.

ml5.js [9] is a JavaScript open source library that aims to make machine learning in the browser approachable for the broader audience. It is a friendly high-level interface to TensorFlow.js, with no external dependencies. This library provides direct access to machine learning models, algorithms as well as pre-trained deep learning models such as YOLO and PoseNet. It also supports the defining, training, and the running of various type of networks such as DNNs, CNNs and RNNs. It provides access to pre-trained models related to applications including object detection, text creation, music composition and more. Ml5.js runs on top of TensorFlow.js and makes functional additions so that some processes are more accessible and easier for the developer, since TensorFlow.js provides a lower level of functionality. A typical example is that the developer does not have to bother about memory management or complicated mathematical operations. Regarding developer support it provides documentation as well as demos and online tutorials.

From all the libraries and frameworks examined, it appears that Tensorflow.js is the most robust framework as it offers the most support for deep learning purposes. The main reason is that it combines both low level programming, such as support and management of many mathematical operations and data manipulation, as well as higher level, such as the creation, training and execution of deep learning models in the browser. It also offers the largest number of supported layer types, activation functions, optimizers as well as network types as it is shown in Table II. Scalability and interconnection with other equally known deep learning frameworks in other programming languages is also a key characteristic. WebDNN is a promising deep learning framework for the inference phase as it offers acceleration via WebGPU technology. The rest of the libraries support fewer features than TensorFlow.js. Consequently, some of them may not be able to solve complicated problems. Libraries such as Brain.js, Synaptic.js and ml5.js can be leveraged for easier but notable problems using deep learning techniques. A major advantage is that they are constantly maintained and developed. Therefore, they are more suitable for a scientific audience that is in an early stage in the field of machine learning.

TABLE II. FRAMEWORK COMPARISON

Features	**Tensor Flow.js**	**Keras. js**	**Conv NetJS**	**brain. js**	**WebDN N**	**Synaptic. js**
Active Status	Yes	No	No	Yes	Yes	Yes
DNN support	Yes	Yes	Yes	Yes	Yes	Yes
RNN support	Yes	Yes	No	Yes	Yes	Yes
CNN support	Yes	Yes	Yes	No	Yes	No
Supported Layer Types	49	-	7	7	-	1
Supported Optimizer Types	7	-	3	1	-	-
Supported Activation Types	16	-	4	4	-	5
Training Support	Yes	No	Yes	Yes	No	Yes
WebGL	Yes	Yes	No	No	Yes	No
WebGPU	No	No	No	No	Yes	No
API-Save model	Yes	No	Yes	Yes	No	Yes
API-Load Model	Yes	Yes	Yes	Yes	Yes	Yes
Import Model from other Frameworks	TF Keras	Keras	-	-	TF Keras Caffe Pytorch	-
Server-side Support	Yes	Yes	Yes	Yes	Yes	Yes
Size (KB)	732	650	33	819	130	106
Documents	Yes	-	Yes	Tutori als	Yes	Yes
Demos	20	9	10	7	8	7

In terms of the suitability of frameworks for defining, training, and running of object detectors in a browser environment, TensorFlow.js appears to be the most suitable. It combines all the features needed for object detection as well as the largest number of components that form an object detector. Also, it provides all the necessary mathematical tools for operations regarding the management of data and tensors for the creation of state-of-the-art object detectors. ConvNet.js is also a powerful framework for the same task, but it lacks the novelty and the supported technologies compared to TensorFlow.js.

IV. PROTOTYPE IMPLEMENTATION

In this section we present a prototype object detection web application. The main purpose of the implementation is the object detection on the client-side, by making use of some of the frameworks mentioned above. Two deep learning models are being used and examined: COCO-SSD and Tiny YOLO. Both deep learning models are pre-trained on COCO dataset. COCO dataset is a large-scale object detection, segmentation and captioning dataset which contains over 200.000 labeled images with 80 object categories and 91 stuff categories. COCO-SSD is provided by TensorFlow.js and TINY YOLO is provided by ml5.js. The application is written in JavaScript. Our model selection has been based on the combination of the size of the models, the classification score that they can achieve and the inference speed. The application can be accessed and used either from a computer or a mobile device, but with different features on each of them due to browser constraints in the webcam access in the mobile devices. The application supports four modes:

1. video detection via webcam or camera access
2. object detection in a snapshot
3. object detection in imported images and

4. file export with the predicted values of the detection.

The application consists of five UIs. The main page of the web application prompts the user to select which of the models they would like to use. Depending on how the user accesses the application and his selection of the underlying model, he is redirected to the corresponding page (see Figure 1).

A. *COCO-SSD*

The first model is COCO-SSD. This model is provided by TensorFlow.js and is a suitable object detector for in-browser deep-learning. This model is based on the SSD architecture and uses a feature extractor based on the MobileNet architecture, called "lite_mobilenet_v2". The main feature of this network is that it can work efficiently on low-resource devices and achieve quite good precision and speed for the given architecture, making use of depthwise separable convolution. In this way, it reduces both the number of parameters and the computational cost. The feature extractor can also be configured with other choices such as "mobilenet_v1" or "mobilenet_v2". For the application we have used the "lite_mobilenet_v2" feature extractor as it has the smallest size and the fastest inference speed. We have also enabled the WebGL backend. The model is pre-trained on COCO dataset and can detect up to 80 different objects.

TensorFlow.js provides all the necessary functions to load, configure and run the model in the browser. Once we have configured our model, we load it on the browser to start the detection of objects for the different modes we have set. We have coded the application in such a way that the loading of the model occurs during the first detection. Consequently, the first detection has a delay of 7-10 seconds. Every other detection in every mode occurs in some milliseconds. For the detection in the snapshot mode and the video detection, we gain access to the user's webcam with the use of MediaDevices.getUserMedia() method. We then mirror the video element to a canvas element, and we feed the canvas element in the model. When the detection finishes, the model returns an object with the predictions. We then manage the predictions and return the results in the corresponding elements so that the user can see the detection. The mode for the detection of imported images works in a similar way. We also have a file export mode with which the user can export all the predictions of every image for every mode in a file.

As for the features of the mobile application we have used an alternate method both for the video detection and the detection of imported images, due to lack of functionality of MediaDevices.getUserMedia() method in some versions of some browsers, such as in iOS Safari 11 and Opera for Android. We have used HTML Media Capture extension to gain access to the user's camera. In this way the user can take a snapshot, import a photo, import a video, take a video, and then make detections.

B. *TINY YOLO*

The second model is TINY YOLO. This model is provided by ml5.js and is an appropriate object detector for in-browser deep-learning, as it has a relatively small size, it has a small number of FLOPS compared to other object detectors and most importantly it uses the YOLO algorithm which is a state-of-the-art method. This specific model can be found as YOLO in ml5.js. It is a YOLO model, but its structure is a TINY YOLO's one. The model is pre-trained on COCO dataset and can detect up to 80 different objects.

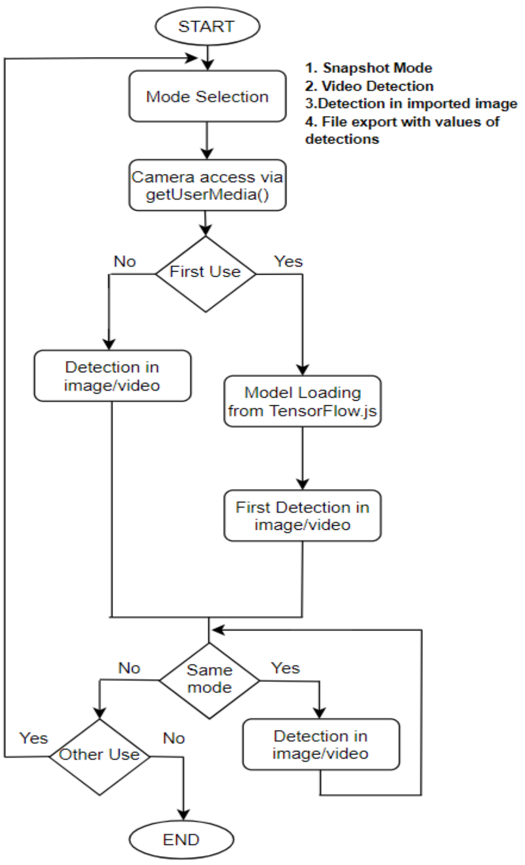
START
Mode Selection
1. Snapshot Mode
2. Video Detection
3.Detection in imported image
4. File export with values of detections
Camera access via getUserMedia()
No
First Use
Yes
Detection in image/video
Model Loading from TensorFlow.js
First Detection in image/video
No
Same mode
Yes
Yes
Other Use
No
Detection in image/video
END

Fig. 1. Workflow of the implementation for COCO-SSD.

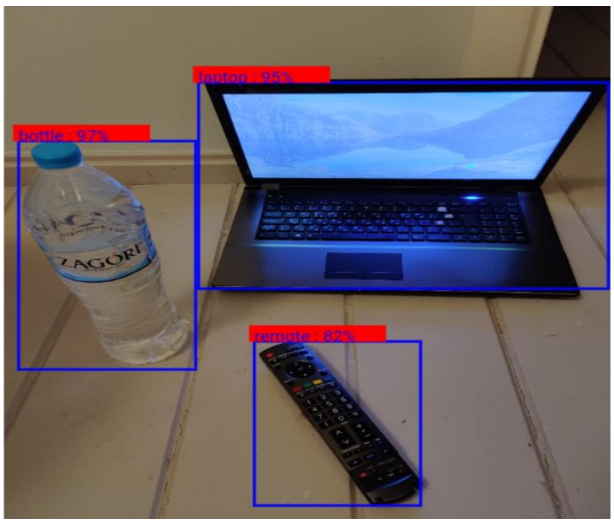

Fig. 2. In-browser object detection with COCO-SSD model.

The library provides all the necessary functions to load, configure and run the model in the browser. Configuration such as Intersection Over Union (IOU) threshold, class probability threshold and bounding box threshold filtering is supported for the model. Once configured, we load the model in the browser to start the detection of objects for the different modes. The modes that the application supports are the same for both models and implemented in a similar manner. Likewise, we have coded the application in such a way that the loading of the model occurs during the first detection. Consequently, the first detection has a delay of 4-6 seconds. Every other detection in every mode occurs in some milliseconds. The mobile application is also implemented accordingly.

V. Evaluation

In this section we present experimental results, concerning the models and the process of object detection in the browser, that emerge from the implementation in Section IV. The results from the implementation confirm that modern models can deal with the problem of object detection in a successful, relatively fast and efficient manner. Our evaluation is carried along the following lines: a) parameters that affect the computational resources consumed, b) inference speed, c) precision of the models and d) concluding observations. All measurements were taken on an ASUS X541UVK laptop with an Intel Core i5-7200U CPU and an Intel HD Graphics 620 processor GPU. The browser that was used for the experiments was Google Chrome.

A. Consumption of Computational Resources

To study the consumption of computational resources an experiment was performed for the CPU and GPU utilization during the inference phase for each mode of the prototype implementation (Section IV). The methodology for the measurements was the following: for the snapshot detection and the imported images we calculated the average utilization in both GPU and CPU during the loading and detection as well as during the detection (after the model is loaded), for 50 images. For the loading of the model and the video detection we used 5 different video streams of 50 seconds duration each. The results of the utilization of both GPU and CPU for each model of the implementation are shown in Table III and IV.

TABLE III. GPU and CPU Utilization for COCO-SSD

Tasks	Average GPU utilization	Average CPU utilization
Snapshot during model loading	4.8%	37.3%
Detection in imported image during model loading	3.9%	35%
Snapshot	23.2%	18.9%
Video detection	79.2%	44.9%
Detection of imported image	22.8%	17.8%

TABLE IV. GPU and CPU Utilization for TINY YOLO

Tasks	Average GPU utilization	Average CPU utilization
Snapshot during model loading	11.3%	36.8%
Detection in imported image during model loading	9.6%	31%
Snapshot	36.1%	19.1%
Video detection	77.7%	34.5%
Detection of imported image	21.5%	10.8%

In comparison, the two models for their loading as well as for the inference phase consume approximately the same resources. TINY YOLO utilizes more GPU and slightly less CPU compared to COCO-SSD. For video detection, COCO-SSD utilizes a little bit more GPU. A possible explanation is that COCO-SSD makes more predictions compared to TINY YOLO for the same time-period. Hence more graphic rendering occurs. As a result, more use of the GPU is being made. In general, the utilization of both GPU and CPU is normal, except for the video detection mode where the utilization is a bit high.

B. Inference Time

An essential parameter that measures how fast the models are and what speed they achieve in the browser environment is the inference time. For this parameter, measurements were taken for the following six tasks: 1) model loading and detection in snapshot mode, 2) model loading and detection of an imported image, 3) model loading and detection of a video, 4) detection of an imported image, 5) detection in snapshot mode and 6) video detection. For the model loading and the detection tasks, the interval between the load initiation and predictions rendering is measured. For detection tasks only the predictions rendering is measured. For each task, except the video detection, we make 50 predictions and we calculate the average inference time of those. The results of the inference time for both models of the implementation for each task, except for the video detection, are shown in Table V.

TABLE V. Inference Time for COCO-SSD and TINY YOLO

Tasks	TINY YOLO	COCO-SSD
Model loading and detection of snapshot	4832 ms	8413 ms
Model loading and detection of imported image	4910 ms	8820 ms
Model loading and video detection	6230 ms	10193 ms
Snapshot detection	685 ms	131 ms
Detection of imported image	475 ms	124 ms

As shown in Table V, the loading time of TINY YOLO for every task is almost 2x faster than COCO-SSD's. However, after the model has been loaded the inference time of COCO-SSD is significantly smaller compared to TINY YOLO's. More specifically for the tasks of snapshot detection and the detection of an imported image, COCO-SSD is almost 4x faster than TINY YOLO. An explanation for this fact is that COCO-SSD is a low latency model that aims in mobile applications as it uses depthwise separable convolution, which reduces the number of parameters, increasing in this way the inference speed. In general, the inference time of both object detectors is relatively fast taking into consideration the resources that are being exploited and the browser environment.

For the video detection task, we are interested in how many detections the models achieve every second. In this way we can understand in how many FPS they operate. This is important because an object detector must be working in high inference speed while being accurate. Many applications make use of videos, and that means that object detectors must be able to render predictions in a decent FPS rate. So, the measurements for this function have been taken in a slightly different way. For the inference time of video detection, measurements were taken over a period of 10 seconds to determine the number of detections that emerged in this period. This process was performed 30 times and the average was calculated to determine in how many FPS the object detectors work. The results are shown in Figure 3.

COCO-SSD achieves higher FPS compared to TINY YOLO. It achieves 8.8 detections per second in average compared to TINY YOLO's 4.6. This is reasonable since the inference time of the predictions of COCO-SSD is significantly smaller to TINY YOLO's, once the model is loaded.

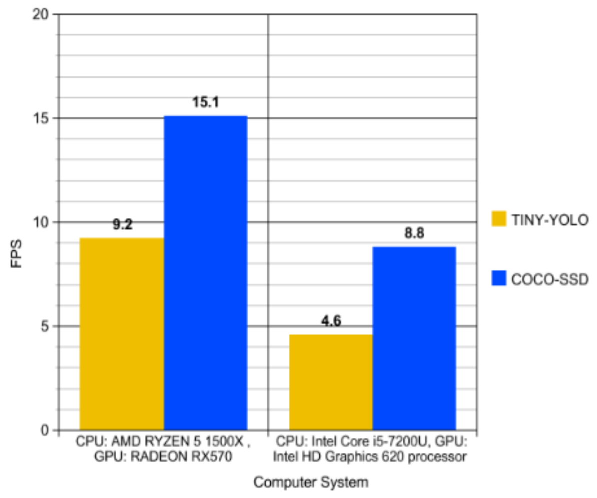

Fig. 3. Inference speed for COCO-SSD and TINY YOLO in video detection.

When the application runs in a different computer system that consists of an AMD Ryzen5 1500X CPU and an AMD Radeon RX570 GPU the inference time is reduced, and more predictions are produced in the same period of testing. Both models boost their speed performance since the FPS of both are almost doubled. Consequently, depending on the computer system resources, the models can achieve better or worse performance in terms of speed.

Compared to Table I, both models have lower inference speed, for two main reasons. Firstly, the models are executed in a browser environment with limited computer resources and not on a standalone powerful GPU. Secondly, TensorFlow.js is almost 2x slower than native TensorFlow even when run on GPU: JavaScript is single-threaded, and the memory allocated for WebGL is limited by the browser [15].

C. Precision

The most basic parameter that determines the accuracy of deep-learning models is precision. For an object detector to be effective, just being functional is not enough, it must be precise as well. Particularly, it must be able to classify and locate the objects as precisely as possible. There is a trade-off between the speed and the accuracy of object detectors [12]. Object detectors with small number of parameters such as SSD with MobileNet feature extractors tend to have smaller inference time but with a cost in the mAP performance compared to object detectors with a big number of parameters such as a Faster-RCNN with an Inception Resnet V2 feature extractor [12]. The object detectors employed in our prototype tend to have a smaller size and a smaller number of parameters compared to other object detectors. However, they try to use techniques to bridge this gap.

In this subsection we evaluate the precision of the object detectors that are used in our prototype. Both models are pre-trained models on the COCO dataset. We evaluate the precision of the models based on a custom dataset that consists of 100 images contained in the COCO dataset. We also evaluate the precision of both models by using the PASCAL VOC2007 metric [5]. According to this specific metric the mean average precision is computed by averaging the average precision of each class from the dataset PASCAL VOC2007 with an IOU fixed at 0.5. We are using an IOU fixed at 0.5 and we calculate the average precision of 8 classes from our 100-image dataset for each model.

TABLE VI. AVERAGE PRECISION OF COCO-SSD AND TINY YOLO WITH FIXED IOU AT 0.5, USING PASCAL VOC2007 METRIC IN OUR CUSTOM DATASET

Categories	COCO-SSD	TINY YOLO
Person	81.2%	58.3%
Car	74.2%	48.4%
Dog	63.3%	46.7%
Cat	81.0%	54.1%
Laptop	80.6%	56.7%
Mouse	64.2%	24.7%
TV	65.0%	73.1%
Keyboard	58.2%	52.5%

As shown in Table VI, COCO-SSD achieves a better average precision for almost every class. Both object detectors have a relatively high precision taking into consideration the speed performance they can achieve. Both object detectors are precise and fast in applications in which a high inference speed is important. Given their inference speed, both detectors achieve a competent precision in detecting objects on the browser environment. Compared to other well-known object detectors, such as Faster-CNN, they have a lower precision due to the smaller number of parameters. However, they achieve a significantly higher inference speed. Overall, the combination of both the precision and their speed makes them suitable for many computer vision applications.

D. Discussion

For identical images with different levels of resolution, which contain many objects, both models performed better detection at the images with the higher resolution, in the sense that they located more objects and their classification score was higher. For images with higher resolution, smaller objects were detected with higher precision compared to images with lower resolution where the objects sometimes were not detected at all. For images that contained one or two objects TINY YOLO performed better at those who had higher resolution and COCO-SSD performed almost with the same precision for both levels of resolution.

Another interesting observation is that COCO-SSD performed better at the detection of bigger objects for the same images. Particularly, the bounding box predictions are more precise as well as the classification task. TINY YOLO performed slightly better at the detection of objects located in the background of the image, such as persons in paintings or books.

Another factor examined was brightness. More specifically, we examined how accurate the models are for three levels of brightness: low brightness, normal brightness, and high brightness. We tested the models with 25 images of different levels of brightness to see how they performed. All images contained between one and three objects. We considered successful a detection with at least one object correctly detected in an image. As shown in Figure 3, COCO-SSD's performance is not affected by the different levels of brightness. On the contrary, TINY YOLO's performance gets affected by the different levels of brightness.

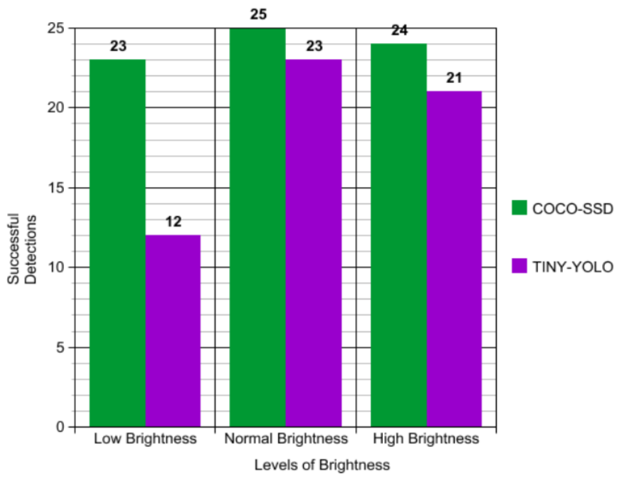

Fig. 4. Performance of COCO-SSD and TINY-YOLO for different levels of brightness.

VI. Conclusions and Future Work

In this paper we have performed a review and a comparative survey over a wide range of frameworks that are suitable for deep-learning purposes in a browser environment. We have also focused on the object detection problem on the client-side and have proposed a prototype web application with several features, which takes advantage of two different object detectors for in-browser object detection while using some of the frameworks examined. We have shown that, by utilizing in-browser deep-learning frameworks it is possible to design, implement and execute deep-learning models solely in a browser, even on mobile devices. Our results verify that object detection can be achieved in a successful and efficient manner despite the relative immaturity of existing frameworks, the limited hardware resources, and other browser constraints.

With the continuous appearance of technologies in deep learning, there appears to be potential in many application fields. For example, an optimized version of our prototype could work efficiently for the creation or improvement of security systems. Feeding the application with CCTV camera images or videos could introduce a “smart” and robust security system. Also, possible advances in object-detector algorithms as well as availability of high-performance hardware can help the application run in higher FPS. Therefore, it can be made suitable for autonomous cars use. Since the image is the main source of information for the detection of signs, crossings and traffic lights, the use of real-time object detectors could help prevent accidents. The application could also work for bigger IoT systems that need a fast and relatively accurate object detection component.

References

[1] Alippi, Cesare, Simone Disabato, and Manuel Roveri. "Moving convolutional neural networks to embedded systems: the alexnet and VGG-16 case." In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 212-223. IEEE, 2018.

[2] Brain.js. “Neural Networks in JavaScript.” https://brain.js.org/.

[3] ConvNetJS. “ConvNetJS: Deep Learning in your browser”. https://cs.stanford.edu/people/karpathy/convnetjs/.

[4] Dakkak, Abdul, Carl Pearson, and Wen-mei Hwu. "Webgpu: A scalable online development platform for gpu programming courses." In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 942-949. IEEE, 2016.

[5] Everingham, Mark, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. "The PASCAL visual object classes challenge 2007 (VOC2007) results." (2007)

[6] Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580-587. 2014.

[7] Girshick, Ross. "Fast r-cnn." In *Proceedings of the IEEE international conference on computer vision*, pp. 1440-1448. 2015.

[8] GitHub. “Keras.js.” https://github.com/transcranial/keras-js.

[9] GitHub. “Synaptic.js.” https://github.com/cazala/synaptic.

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

[11] Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

[12] Huang, Jonathan, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer et al. "Speed/accuracy trade-offs for modern convolutional object detectors." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7310-7311. 2017.

[13] Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. "Microsoft coco: Common objects in context." In *European conference on computer vision*, pp. 740-755. Springer, Cham, 2014.

[14] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. "Ssd: Single shot multibox detector." In *European conference on computer vision*, pp. 21-37. Springer, Cham, 2016.

[15] Ma, Yun, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. "Moving deep learning into web browser: How far can we go?." In *The World Wide Web Conference*, pp. 1234-1244. 2019.

[16] MIL-Tokyo. “MIL WebDNN.” https://mil-tokyo.github.io/webdnn/.

[17] Ml5js. “ml5js·Friendly Machine Learning For The Web.” https://ml5js.org/.

[18] Parisi, Tony. *WebGL: up and running*. " O'Reilly Media, Inc.", 2012.

[19] Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." *arXiv preprint arXiv:1804.02767* (2018).

[20] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779-788. 2016.

[21] Redmon, Joseph. YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolo/.

[22] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks." In *Advances in neural information processing systems*, pp. 91-99. 2015.

[23] Szegedy, Christian, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. "Inception-v4, inception-resnet and the impact of residual connections on learning." In *Thirty-first AAAI conference on artificial intelligence*. 2017.

[24] TensorFlow. “TensorFlow.js: Machine Learning for Javascript Developers.” https://www.tensorflow.org/js.