

Minh Khanh Pham

**DEVELOPMENT OF CHROME EXTENSION FOR
REAL-TIME YOLO-BASED OBJECT
DETECTION USING TENSORFLOW.JS**

Bachelor's Thesis
Faculty of Engineering and Natural Sciences
Examiner: Jussi Kalliola
April 2024

ABSTRACT

Minh Khanh Pham: Development of Chrome Extension for real-time YOLO-based object detection using TensorFlow.js
Bachelor's Thesis
Tampere University
Science and Engineering
April 2024

This thesis investigates how deep learning models can be efficiently deployed in client-side web browser environments through a practical implementation of a real-time object detection system. The project demonstrates this by converting and integrating a YOLO11n model into a Chrome Extension using TensorFlow.js. The extension performs all inference locally in the browser using WebGL acceleration, allowing for real-time object detection without sending data to external servers. Key features include keyword-based filtering, visual overlays, and adaptive performance management. The system was evaluated across different hardware configurations, and was compared with a related web-based application. The results support the feasibility of browser-based deep learning as a privacy-preserving alternative to server-side inference.

Keywords: browser-based machine learning, object detection, TensorFlow.js, web extension, front-end deep learning

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

PREFACE

This thesis was completed as part of my Bachelor's degree in Science and Engineering at Tampere University. I would like to express my sincere gratitude to my supervisor, Jussi Kalliola, for his valuable feedback and continuous support throughout the process. I also thank my friends and peers who participated in testing the application on different systems and offered useful insights for improvement.

This work would not have been possible without the open-source community, whose tools and documentation provided a solid foundation for experimentation and implementation.

Tampere, 30th April 2024

Minh Khanh Pham

CONTENTS

1. Introduction	1
2. Background	3
2.1 Object detection in web browsers	3
2.2 TensorFlow.js for browser-based machine learning	4
2.3 Overview of YOLO	5
2.4 Overview of Chrome Extensions	7
3. Related work	8
4. System architecture and design	9
4.1 Overview of the system architecture	9
4.2 Core system components	10
5. Implementation	12
5.1 Exporting YOLO11n model to TensorFlow.js format	12
5.2 Application structure and development	13
5.3 User interface	14
5.4 Detection logic	15
5.5 Object detection pipeline	16
5.6 Object tracking and keyword filtering	17
5.7 Resource and performance management	18
6. Results and discussion	20
6.1 Performance results	20
6.2 Security and privacy challenges and solutions	22
6.3 Limitations and potential improvements	23
7. Conclusion	24
References	25

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence, the simulation of human intelligence processes by machines, especially computer systems
API	Application Programming Interface
C2PSA	Channel and Spatial Attention Module with 2 convolutions
C3k2	Convolutional block with 3 convolutions and kernel size 2
CNN	Convolutional Neural Networks
COCO	Common Objects in Context, a large-scale object detection, segmentation, and captioning dataset
CPU	Central Processing Unit
CSP	Cross Stage Partial
CSS	Cascading Style Sheets, used for describing the look and formatting of a document written in HTML
DOM	Document Object Model, a programming interface for web documents used to manipulate HTML and XML content
E-ELAN	Enhanced Efficient Layer Aggregation Network
ELAN	Efficient Layer Aggregation Network
FPS	Frames Per Second, a measurement of how many images (frames) are processed/rendered each second
GANs	Generative Adversarial Networks
GELAN	Generalized ELAN
GHz	Gigahertz, a unit of frequency equal to one billion cycles per second
GPU	Graphics Processing Unit
HTML	HyperText Markup Language, the standard markup language for creating web pages
HTTPS	Hypertext Transfer Protocol Secure, an extension of HTTP that uses encryption (SSL/TLS) to secure communication over a computer network, widely used on the Internet

i5-1135G7	Intel Core i5 11th Gen mobile CPU model designed for lightweight laptops
i5-11400H	Intel Core i5 11th Gen high-performance CPU model for gaming and professional laptops
ID	Identifier, often used to uniquely represent an object, element, or user
IEEE	Institute of Electrical and Electronics Engineers
IoU	Intersection over Union, a metric used to evaluate the overlap between predicted and ground truth bounding boxes
JS	JavaScript, a programming language commonly used for creating interactive effects within web browsers
kB	Kilobyte, a unit of digital information equal to 1,024 bytes
ms	Milliseconds, a unit of time equal to one thousandth of a second
NMS	Non-Maximum Suppression
OD	Object Detection
ONNX	Open Neural Network Exchange, an open format for representing machine learning models, supported by many frameworks
PANet	Path Aggregation Network
PGI	Parallel Guided Information
RAM	Random Access Memory, a type of volatile memory used by computers for fast data access
R-CNN	Region-based Convolutional Neural Networks
SPPF	Spatial Pyramid Pooling – Fast variant
SPP	Spatial Pyramid Pooling
TF.js	TensorFlow.js
UI	User Interface
URL	Uniform Resource Locator
WebGL	Web Graphics Library
WSL	Windows Subsystem for Linux, a compatibility layer for running Linux binary executables natively on Windows
YOLO	You Only Look Once, a machine learning model

1. INTRODUCTION

In the era of web-based technologies and interactive applications, the integration of artificial intelligence and machine learning into client-side environments [1][2] has become increasingly feasible and desirable. One such advancement is the deployment of deep learning models within web browsers [1][2], eliminating the reliance on cloud-based services and enhancing the user experience with faster inference, greater privacy, and offline capabilities.

Object detection [3], a key task in computer vision, involves identifying and localizing multiple objects within an image or video frame. Traditionally, such tasks require powerful servers or back-end systems due to the computational cost associated with model inference. However, recent developments in lightweight neural network architectures like YOLO (You Only Look Once) and browser-based machine learning libraries such as TensorFlow.js [4] have made it possible to perform complex tasks like object detection directly in the browser using JavaScript.

The research question addressed in this thesis is: How can deep learning models be efficiently deployed in client-side browser environments, and how can this approach be demonstrated using an object detection system? To answer this research question, the thesis explores the deployment of deep learning models in browser environments by converting and optimizing a YOLO model for TensorFlow.js, integrating it into a Chrome Extension, and evaluating the resulting system in terms of real-time performance, responsiveness, and usability. This deployment approach provides significant advantages: it enables integration with the user's browser environment, requires no installation of additional software or dependencies, and allows direct interaction with web content using Chrome APIs [1]. Chrome Extensions [5][6] provide direct access to browser content, offer seamless UI integration, and support script injection and screen capture via secure APIs. Compared to traditional web apps, they simplify distribution and enhance privacy by enabling full client-side execution [7].

This thesis is structured as follows. Chapter 2 provides the theoretical foundations for deploying deep learning in browsers, covering object detection fundamentals, Chrome extension architecture, and the capabilities of TensorFlow.js. Chapter 3 reviews related work, comparing existing browser-based implementations. Chapter 4 presents the sys-

tem design, including architecture and core components. Chapter 5 details the implementation process, including model export, UI integration, and detection logic. Chapter 6 discusses experimental results, performance comparison with related tools, and observed challenges. Finally, Chapter 7 concludes the study and outlines directions for future work.

2. BACKGROUND

This chapter presents the background required to understand the deployment of deep learning models within web browsers. It begins by introducing the object detection task and the unique challenges and opportunities that arise in browser environments. It then discusses TensorFlow.js, the JavaScript-based machine learning framework used to run inference directly in the browser. The chapter also includes an overview of the YOLO model family and examines how Chrome extensions enable direct integration with browser content and APIs. These foundational topics set the stage for the system design and implementation described in later chapters.

2.1 Object detection in web browsers

Object detection [3] is a computer vision task to identify and locate objects in videos and images. Traditionally, object detection models [7] were run on servers due to their computational complexity. However, advancements in web-based machine learning frameworks, such as TensorFlow.js, have enabled the deployment of these models directly in the browser [1][7].

Running object detection in the browser offers several advantages [1][7]. One of the primary advantages of is that it keeps user data entirely on the client side, avoiding the need to send potentially sensitive information to remote servers. The local inference model significantly reduces latency, allowing for real-time feedback. Additionally, the ability to operate offline makes browser-based solutions robust in environments with limited or unstable internet connectivity. Any device with a compatible browser can run the application, and updates can be carried out without requiring explicit installation or downloads.

Despite these advantages, browser-based object detection also presents several challenges [1][7]. Resource limitations in the browser environment constrain the size and complexity of models that can be deployed. While frameworks like TensorFlow.js leverage WebGL to access hardware acceleration, the performance still falls short compared to native GPU-based environments. Furthermore, the open nature of client-side code introduces concerns about model security, since models can be easily inspected or extracted by end users. These applications are typically optimized for low-power devices and simple use cases; complex workloads may result in higher latency or a slow user ex-

perience. However, the growing maturity of browser-based machine learning tools makes this deployment model increasingly viable for lightweight, privacy-sensitive, and interactive applications.

2.2 TensorFlow.js for browser-based machine learning

TensorFlow.js (TF.js) is an open-source JavaScript library designed for running machine learning models directly in web browsers or Node.js environments [1]. It enables in-browser training and inference, eliminating the need for server-side computations and allowing for enhanced user privacy, reduced latency, and improved performance in real-time applications [2].

JavaScript is currently the most popular programming language according to StackOverflow Developer Survey 2024 [8] and the third most popular programming language according to IEEE Spectrum 2024 [9]. The widespread use has made JavaScript a natural environment for deploying machine learning applications, particularly in the browser [7]. Libraries like TensorFlow.js lower the barrier for developers without a Python or C++ background to build and experiment with machine learning models directly in JavaScript. This approach of machine learning encourages innovation enables a broader set of users to create interactive, privacy-preserving, on-device applications.

One of the key advantages of TensorFlow.js is its ability to work with pre-trained models, including YOLO models converted into TF.js format [2][4]. This conversion process allows developers to transform YOLO models into a browser-compatible format. By leveraging TF.js, developers can integrate YOLO-based object detection into Chrome Extensions and web applications, enabling real-time inference directly within the user's browser.

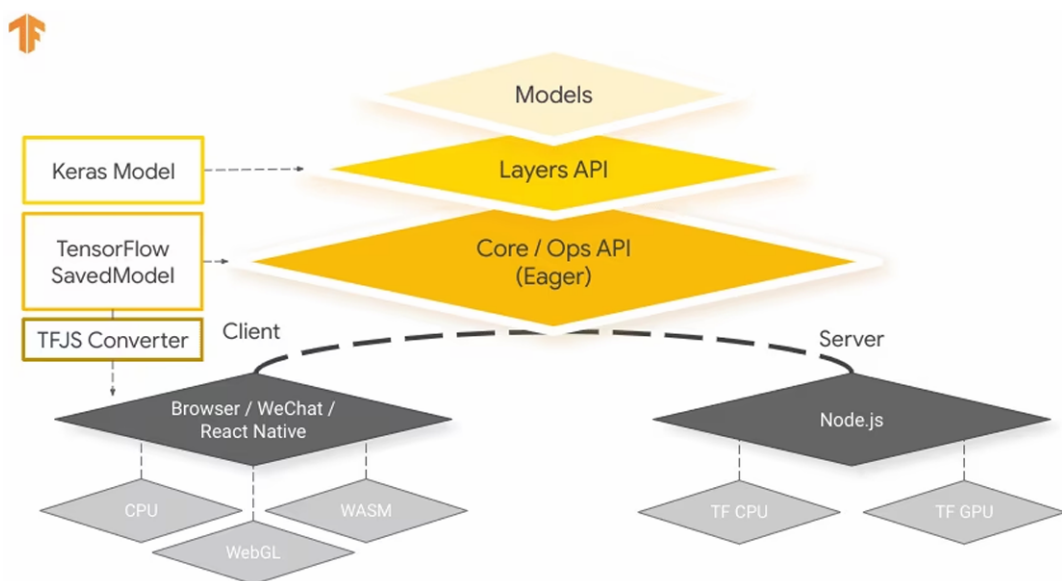


Figure 2.1. Overview of the TensorFlow.js architecture [4]

A visual representation of the TensorFlow.js ecosystem, as shown in Figure 2.1, illustrates how models can be deployed both on the client-side (browser) and server-side (Node.js). The figure demonstrates the workflow, starting from converting a Keras or TensorFlow SavedModel using the TF.js converter, which enables execution in different environments. On the client-side, models can run within browsers, WeChat, or React Native applications, utilizing WebGL for GPU acceleration, WebAssembly for optimized CPU performance, and other execution backends [4] [2]. On the server-side, models can be deployed using Node.js with TensorFlow CPU or GPU support, allowing for scalable machine learning inference. This architecture highlights the flexibility of TensorFlow.js in supporting both local and server-based machine learning applications.

2.3 Overview of YOLO

YOLO (You Only Look Once) is a real-time object detection system that reframes detection as a single regression problem, directly predicting class probabilities and bounding boxes from entire images [3]. Unlike earlier systems such as R-CNN that use region proposals and multistage pipelines, YOLO uses a single convolutional neural network (CNN), making it extremely fast and end-to-end trainable.

The detection pipeline involves three main steps: resizing the input image to a fixed size, passing the image through a CNN, and applying confidence thresholding and non-maximum suppression to refine final predictions [3]. This process is illustrated in Figure 2.2, adapted from the original YOLO paper.

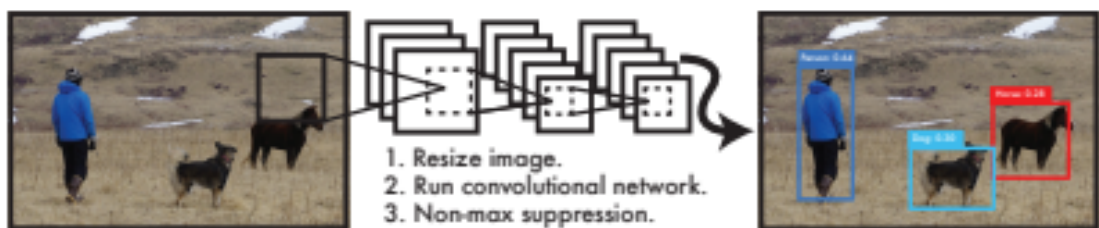


Figure 2.2. YOLO detection system [3]

Over the years, the YOLO framework has undergone significant evolution, improving in accuracy, speed, and versatility [10]. Table 2.1 summarizes major milestones in YOLO's development.

Release	Year	Tasks	Contributions	Framework
YOLO	2015	Object Detection, Basic Classification	Single-stage object detector	Darknet
YOLOv2	2016	Object Detection, Improved Classification	Multi-scale training, dimension clustering	Darknet
YOLOv3	2018	Object Detection, Multi-scale Detection	SPP block, Darknet-53 backbone	Darknet
YOLOv4	2020	Object Detection, Basic Object Tracking	Mish activation, CSPDarknet-53 backbone	Darknet
YOLOv5	2020	Object Detection, Basic Instance Segmentation (via custom modifications)	Anchor-free detection, SWISH activation, PANet	PyTorch
YOLOv6	2022	Object Detection, Instance Segmentation	Self-attention, anchor-free OD	PyTorch
YOLOv7	2022	Object Detection, Object Tracking, Instance Segmentation	Transformers, E-ELAN reparameterisation	PyTorch
YOLOv8	2023	Object Detection, Instance Segmentation, Panoptic Segmentation, Keypoint Estimation	GANs, anchor-free detection	PyTorch
YOLOv9	2024	Object Detection, Instance Segmentation	PGI and GELAN	PyTorch
YOLOv10	2024	Object Detection	Consistent dual assignments for NMS-free training	PyTorch
YOLOv11	2024	Object Detection, Instance Segmentation, Pose Estimation, Image Classification	C3k2 blocks, SPPF, C2PSA attention	PyTorch

Table 2.1. Evolution of YOLO models [10]

YOLOv11 represents the most recent development in the YOLO series, introduced in 2024 to improve real-time performance, flexibility, and task diversity [10]. It features enhance-

ments in its backbone and detection heads, including C3k2 blocks, SPPF, and C2PSA attention modules, which improve feature extraction and spatial reasoning. The model supports a wide range of computer vision tasks, such as object detection, instance segmentation, pose estimation, and image classification, and is available in multiple sizes for deployment across edge and cloud devices. YOLOv11 maintains high accuracy while offering better inference speed and model scalability. Regarding deployment, while YOLO is traditionally trained and run using frameworks like PyTorch or TensorFlow (Python), newer tools now support export to TensorFlow.js [4]. This allows developers to execute YOLO models directly in the browser using JavaScript and WebGL.

2.4 Overview of Chrome Extensions

Chrome extensions are small software programs that customize the browsing experience in the Google Chrome browser [5]. They are built using standard web technologies such as HTML, CSS, and JavaScript, and they interact with web pages and browser features through the Chrome Extensions API [6]. Extensions can modify the appearance of web pages, add new functionality, or interact with external services. The architecture of a Chrome extension typically includes a manifest file which defines the extension's metadata and permissions, background service workers for long-running processes, content scripts to interact with web pages, and a user interface such as popups or options pages that run in their respective browser processes.

Chrome extensions are executed as part of the browser environment, not as standalone external applications [6]. Specifically, each extension runs in an isolated context within Chrome's multi-process architecture, sharing system resources with the browser while maintaining sandboxed execution for security. This architecture allows Chrome Extensions to behave like modular web applications with dedicated access to browser internals while respecting modern browser security policies and user privacy. Therefore, Chrome Extensions offer a powerful and secure way to deploy interactive functionalities, such as client-side object detection, directly into users' browsers.

3. RELATED WORK

TensorFlow.js is introduced as part of the broader TensorFlow ecosystem and demonstrated its capability to perform both training and inference in client-side applications using WebGL for acceleration [7]. This work laid the foundation for the integration of deep learning tasks, such as classification, pose estimation, and object detection, within browser environments.

A few public projects have used TensorFlow.js in object recognition tasks. For instance, the official TensorFlow.js Chrome extension example demonstrates a basic pipeline for loading a pre-trained model, and performing image classification [11]. However, this implementation focuses on classification tasks using models like MobileNet, which lack the spatial precision and complexity of object detection models such as YOLO.

In the object detection domain, a notable open-source work is the TensorFlow.js implementation of YOLOv8 by Hyuto [12], which provides tools for converting YOLO models to TensorFlow.js format and running them within a browser-based interface. This project showcases the feasibility of deploying high-performance object detection models entirely in the browser, but it does not integrate with the Chrome extension platform or support real-time interaction with the browser's content.

Another relevant implementation is a publicly available Chrome Extension titled Video Object Detection, which allows users to detect objects in videos displayed on web pages [13]. While the exact detection model used is not disclosed by the developer, the extension demonstrates how client-side inference can be used for real-time visual overlays in the browser. However, its functionality appears to be limited in class variety.

Although these previous efforts highlight the progress in browser-based object detection, none have combined a YOLO-based model with the Chrome extension platform to enable real-time detection of arbitrary browser content using screen capture and Tensor-Flow.js. This thesis addresses this gap by implementing a Chrome Extension that performs object detection using a YOLOv11 lightweight model converted to TensorFlow.js, offering a privacy-respecting, responsive, and install-free solution.

4. SYSTEM ARCHITECTURE AND DESIGN

This chapter presents the architecture and design principles that guided the development of the object detection Chrome Extension. The system was structured to support real-time performance, client-side execution, modular scalability, and privacy-focused operation. The implementation uses JavaScript modules, Chrome extension APIs, and TensorFlow.js to run object detection directly within the browser without requiring external servers or background processing.

4.1 Overview of the system architecture

The architecture of the object detection system, as shown in Figure 4.1 is designed as a layered, modular client-side application. The system is organized into four primary layers: user interface, detection logic, model management, and browser integration. These layers work together to ensure a clean separation of concerns, allowing each part of the system to operate independently while collaborating through standardized data flows.

At the highest level, the system begins with user interaction via a popup interface, which allows users to input detection preferences and start a detection session. Once detection is initiated, the extension opens a dedicated detection tab where screen content is captured and processed through the object detection pipeline. The YOLO11n model is loaded and executed using TensorFlow.js, with real-time results rendered back to the user as bounding boxes and labels. Each layer of the architecture plays a critical role in coordinating this sequence, managing resources, and maintaining performance.

The architecture ensures that all inference and data processing take place locally within the user's browser. This design not only enhances real-time responsiveness but also addresses growing concerns around data privacy by preventing any form of data transmission to external servers. The system also adheres to Chrome's security model through Manifest V3, ensuring a secure and permission-controlled execution environment [6].

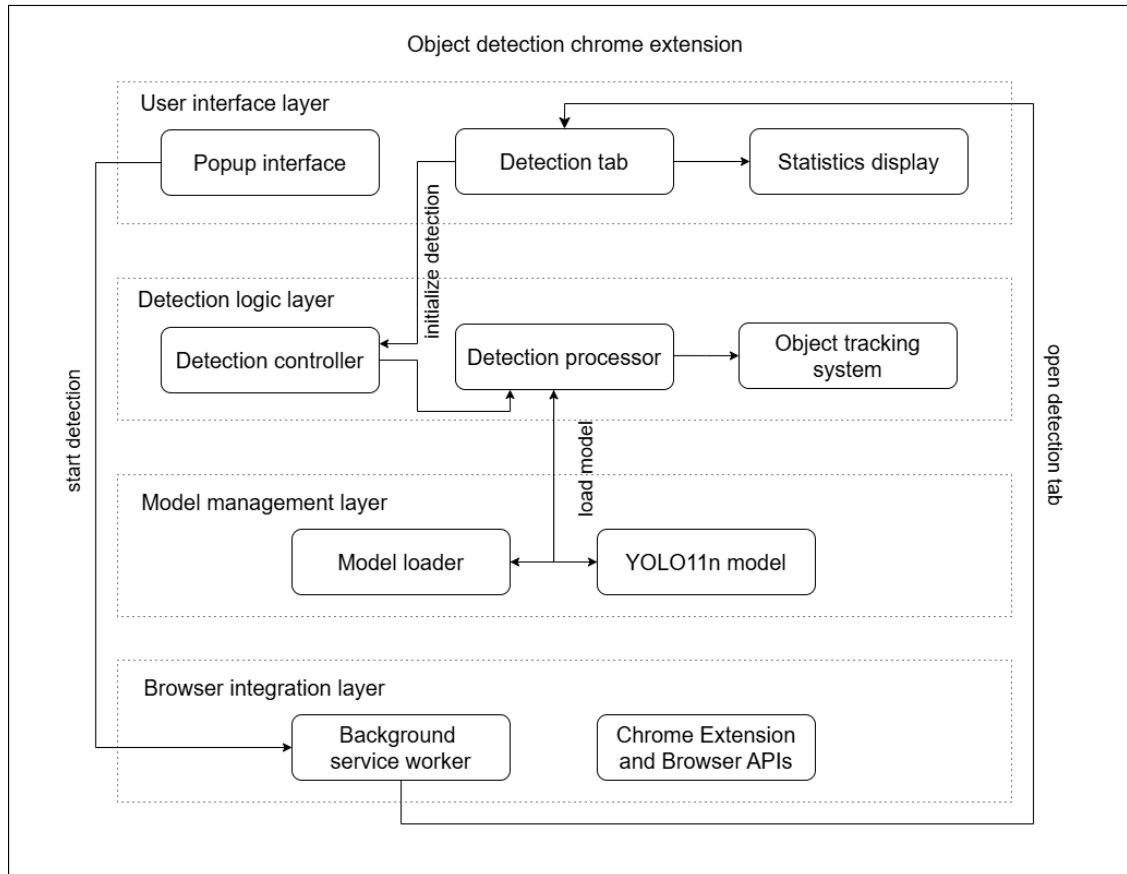


Figure 4.1. Overview of the system architecture

4.2 Core system components

The extension is implemented as a set of cohesive modules, each corresponding to a logical responsibility in the detection system. These modules are grouped within the architectural layers described earlier.

Within the user interface layer, the popup interface serves as the entry point to the system. It allows users to enter a keyword representing the object they wish to detect, such as "person" or "car". Once submitted, the keyword is stored using the Chrome storage API and passed to the background service worker. This interaction triggers the "start detection" sequence. The service worker either opens a new detection tab or focuses on an existing one, where the detection interface is loaded.

The detection interface presents a live video display along with overlay visuals such as bounding boxes and object labels. It contains the user controls to begin or pause detection, and a statistics panel displays real-time updates of detected objects. After the interface loads, it sends an "initialize detection" message to the detection controller to begin the detection session. The detection logic layer handles the runtime operations of the system. The detection controller manages the lifecycle of detection. Captured video

frames are passed to the detection processor, where they are resized, normalized, and prepared for inference. The processor requests a loaded model from the model loader via the “load model” flow and then performs inference using TensorFlow.js. Once predictions are returned, the processor filters overlapping boxes using NMS and matches object classes against the user’s keyword. Object tracking is also applied using a lightweight matching algorithm to maintain consistent object identity across frames. The results are passed back to the user interface, where they are rendered with smooth transitions.

The model management layer manages everything related to model loading and inference execution. It includes the model loader, which is responsible for loading the YOLO11n model and parsing its metadata for use during classification. The YOLO11n model, converted to a TensorFlow.js-compatible format, is optimized for in-browser performance with WebGL acceleration and supports detection of 80 common object classes. Once loaded, the model remains cached and is reused across detection frames for performance.

The browser integration layer enables the extension to communicate with the Chrome runtime environment. The service worker listens for messages from the popup and opens the detection tab when requested. The application uses Chrome Extension APIs for and tab creation (`chrome.tabs.create`), storage management (`chrome.storage.local`), and cross-component messaging (`chrome.runtime.sendMessage`). It also uses standard browser APIs for screen capture (`navigator.mediaDevices.getDisplayMedia`), visibility management (`document.visibilitychange`), and the animation frame loop (`requestAnimationFrame`). Together, these APIs enable the extension to capture screen content, efficiently process frames, and create a responsive user experience that adapts to browser state changes while respecting system resources and user permissions.

Data flows through the system in structured stages. After user input, data is passed from the popup to the service worker, then to the detection interface. From there, detection is initialized, screen capture begins, and video frames are streamed into the detection processor. Inference results from the YOLO model are then transformed and rendered visually. At every step, the architecture ensures synchronization between modules while minimizing interdependencies through clean interface boundaries.

5. IMPLEMENTATION

This chapter presents the implementation details of the Chrome extension for real-time object detection. The development process adheres to the layered architecture described in chapter 3 and is structured to support performance optimization, privacy preservation, and modular design. The implementation includes model preparation, user interface construction, detection logic orchestration, and performance management. The system is built entirely in JavaScript and executes inference using TensorFlow.js in the browser. Each section in this chapter describes the necessary steps and technical components involved in bringing the object detection extension into operation. The full implementation is available in the project's GitHub repository ¹.

5.1 Exporting YOLO11n model to TensorFlow.js format

To enable real-time object detection directly within the browser, it was necessary to convert the YOLO11n model from its original PyTorch format into a format compatible with TensorFlow.js. YOLOv11n is the nano variant of Ultralytics' YOLOv11 series, optimized for real-time object detection on resource-constrained environments [14].

While the Ultralytics framework provides direct tools to export YOLO models to various formats, including TensorFlow.js, a technical limitation had to be addressed during implementation. Specifically, TensorFlow Decision Forests, a dependency required by some conversion pipelines, is not currently supported on native Windows platforms. According to the official documentation, the package is not available as a pip installation on Windows and must instead be used through Docker or within a Linux environment [15]. To overcome this limitation, the model conversion process was carried out inside a Windows Subsystem for Linux (WSL) environment running Ubuntu. This provided access to the full set of TensorFlow and Ultralytics tools while preserving compatibility with the host development system.

Within the WSL Ubuntu shell, the Python-based Ultralytics library was installed using the pip package manager. After the YOLO11n model was downloaded, the export was performed using either a Python script or the command-line interface [4]. Upon successful export, the process generated a directory named `yolo11n_web_model` which contains

¹<https://github.com/khanhpham2134/object-detection-chrome-extension>

the converted TensorFlow.js model structure and its associated weight shards. The output includes a `model.json` file that defines the model graph, a series of `.bin` files that store the model weights in a compressed format, and a metadata file that maps object class labels based on the COCO dataset.

This exported model is used within the detection logic of the Chrome extension. It is loaded using TensorFlow.js APIs and executed in the browser using WebGL acceleration. This approach provides several practical advantages, including improved responsiveness, client-side privacy, and the ability to run the extension in offline or constrained environments.

5.2 Application structure and development

This section outlines the software tools and configurations used throughout the implementation phase. The extension was developed using Node.js and built with Vite, a fast JavaScript bundler optimized for ES modules. The primary development environment was Visual Studio Code running on Windows 11, with Google Chrome serving as the testing platform in developer mode.

TensorFlow.js provided the backend for executing deep learning inference directly in the browser. The application relied on the WebGL backend for real-time GPU acceleration, allowing lightweight deployment without needing native installations.

The Vite build configuration was designed to handle multiple entry points for the popup, detection page, and service worker. To optimize loading performance, TensorFlow.js was split into a separate build chunk and cached by the browser.

During development, live-reloading and hot module replacement were enabled to accelerate testing. The output directory was configured as `dist/`, and the Chrome extension was loaded via `chrome://extensions` in unpacked mode.

The structure of the application is illustrated in Figure 5.1.

```

chrome-extension/
├── detection/                # Object detection logic
│   ├── detection.js        # Entry point for detection page
│   ├── model.js           # Loads and manages the model
│   ├── detect.js          # Detection and tracking logic
│   └── ui.js              # Renders bounding boxes and UI
├── public/                 # Static assets
│   ├── detection.html     # Detection UI HTML
│   ├── popup.html        # Extension popup HTML
│   ├── manifest.json      # Chrome manifest file
│   ├── service-worker.js  # Background worker
│   └── images/           # Icons and graphics
├── yolo11n_web_model/     # Pre-trained model
│   ├── model.json        # Model weights
│   ├── weights.bin       # Model weights
│   └── metadata.yaml     # Model metadata
├── popup.js               # Handles popup interactions
├── vite.config.js        # Build configuration
├── build.js              # Custom build script
└── package.json          # Project dependencies

```

Figure 5.1. Hierarchy structure of the application

5.3 User interface

The user interface design of the object detection Chrome extension follows a modular architecture with two distinct views: a configuration popup and a detection interface. This separation of concerns improves usability and maintainability, allowing each part of the extension to be optimized for its specific purpose.

The popup interface serves as the user's entry point for configuring detection parameters. Its design follows principles of minimalism and clarity, providing a single keyword input field accompanied by a dropdown of commonly detected objects. This allows users to either manually specify a detection target or select from pre-defined suggestions. The logic for handling input and initiating detection is implemented in `popup.js`. When a keyword is entered, it is saved to Chrome's local storage to ensure consistency across sessions. The popup then communicates with a background service worker to open the detection tab.

The service worker (`service-worker.js`) listens to messages from the popup and handles tab creation asynchronously. This decouples interface logic from browser-level ac-

tions and improves performance. The detection view is responsible for presenting object recognition results and managing user interactions during an active detection session. It includes a screen preview, a bounding box overlay, control buttons, and a statistics panel in the upper right.

The interface of the application is illustrated in Figure 5.2.

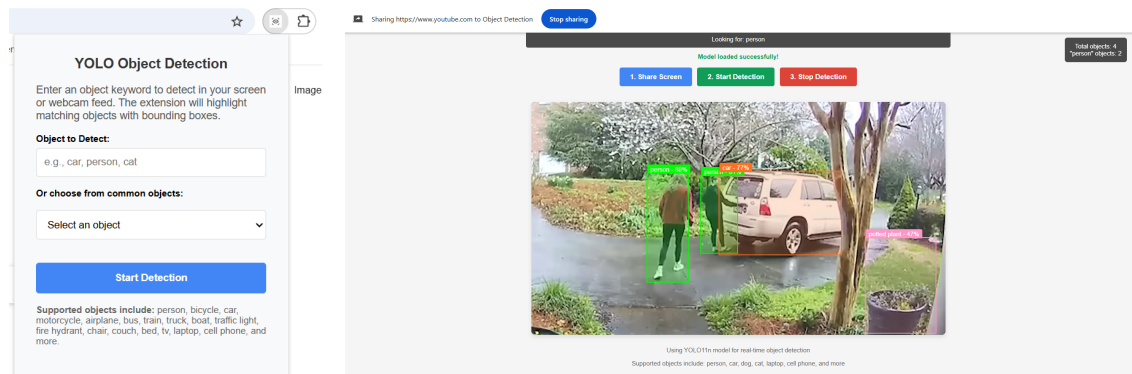


Figure 5.2. *Popup interface (left) and detection interface (right)*

Bounding boxes are rendered using DOM elements rather than canvas. This allows each bounding box to be styled individually and updated selectively. Elements are styled dynamically and updated using CSS transitions. A label repositioning algorithm ensures detection labels remain visible near screen edges. User interactions follow a guided work-flow. Screen sharing must be initiated before detection begins, and button states reflect the detection state.

5.4 Detection logic

The detection logic is triggered when the user interacts with the Chrome Extension interface. Once a keyword is entered and screen capture is authorized, the system initializes core components and begins processing the shared screen content in real time.

The process starts with the `DOMContentLoaded` event, which triggers the initialization of necessary elements such as loading the YOLO model, class metadata, and stored user preferences. When the user clicks the "Share Screen" button, the extension invokes the `navigator.mediaDevices.getDisplayMedia()` API to obtain a media stream, which is then rendered in a video element on the detection interface. When the user starts the detection process by clicking the "Start Detection" button, the system validates all required conditions, checking that a valid keyword has been entered and that a video stream is available. Once confirmed, the interface sets up an overlay layer and enters the main detection loop.

When the user clicks "Stop Detection" the application terminates the loop, removes UI

overlays, resets state variables, and releases TensorFlow resources. This ensures efficient memory usage and prevents potential memory leaks.

The structure of the detection logic is illustrated in 5.3.

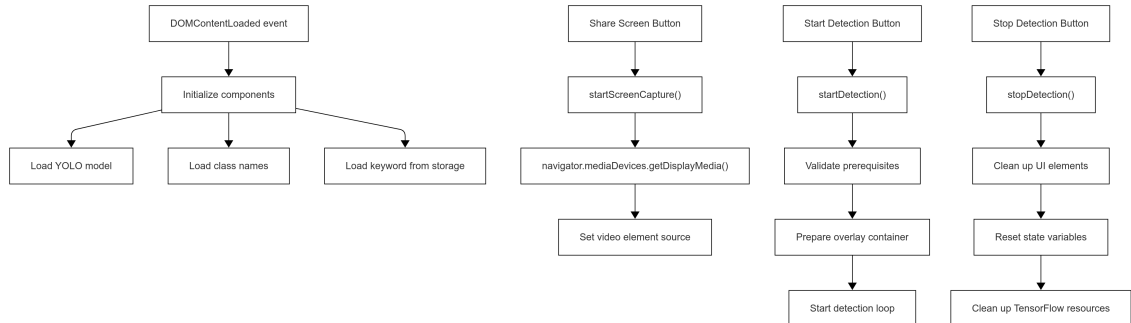


Figure 5.3. Detection logic structure

5.5 Object detection pipeline

The object detection pipeline initiates when the detection loop is active. This loop is managed by `requestAnimationFrame` API, allowing for efficient, frame-synchronized processing. A dynamic throttling mechanism regulates the detection interval based on recent processing time measurements to avoid overloading the browser. The object detection pipeline is illustrated in Figure 5.4.

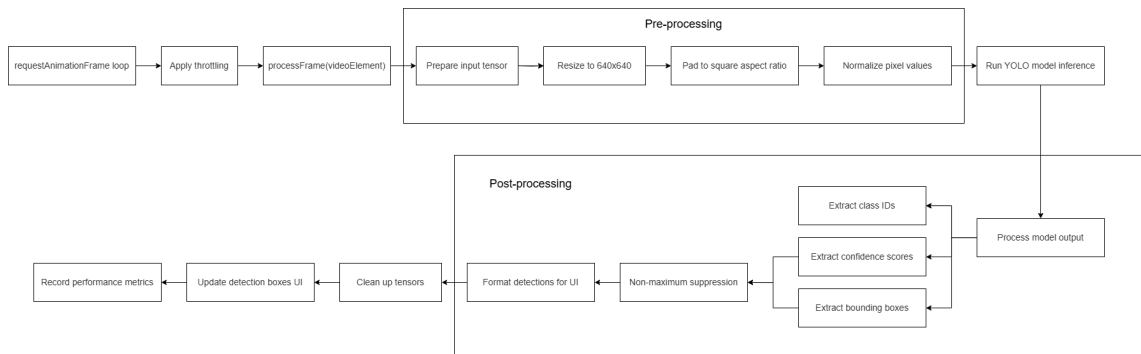


Figure 5.4. Detection pipeline

Each cycle of the loop begins by capturing a frame from the video element, which is passed into the preprocessing phase. Here, the image is resized to 640x640 pixels, padded to maintain its aspect ratio, and normalized to a $[0, 1]$ pixel range to fit the model's expected input.

Following preprocessing, the input tensor is fed into the YOLO11n model using TensorFlow.js, producing raw predictions. These predictions are then post-processed to extract bounding boxes, confidence scores, and class labels. A Non-Maximum Suppression

(NMS) step is applied to reduce overlapping detections using an Intersection-over-Union (IoU) threshold.

In object detection, the IoU [16] is calculated as the ratio between the area of overlap and the area of union of two bounding boxes, as shown in Figure 5.5. A higher IoU value indicates a better overlap between predicted and actual object locations. Non-Maximum Suppression (NMS) [16] is applied as a post-processing step to filter out redundant detections. NMS retains only the bounding box with the highest confidence score while suppressing others with a high IoU overlap. This reduces visual clutter and improves detection accuracy and clarity.

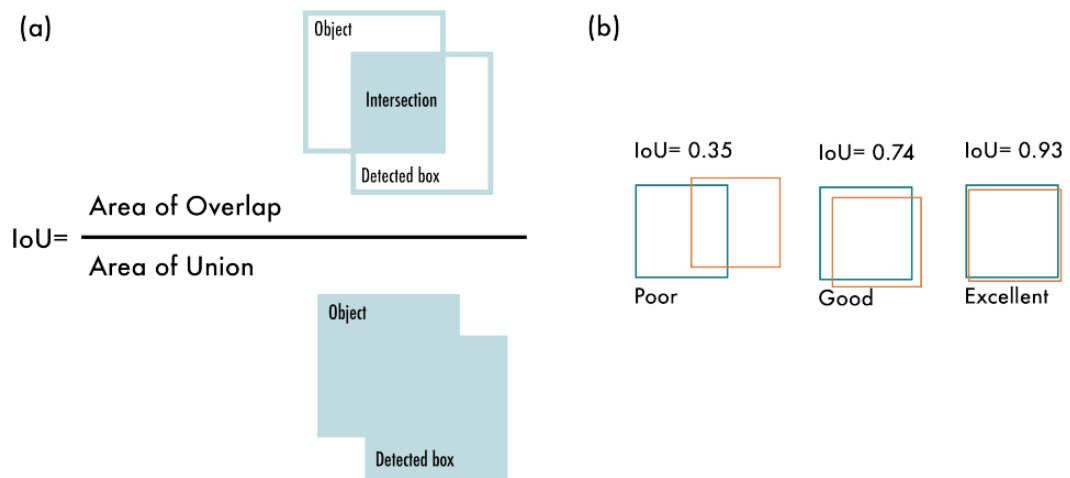


Figure 5.5. (a) Illustration of IoU; (b) examples of IoU values for different box locations [16]

The remaining results are formatted for UI rendering, TensorFlow tensors are disposed to release memory, and the interface is updated with detection overlays. Performance metrics such as frame time and frames-per-second (FPS) are also logged to support the adaptive throttling system.

5.6 Object tracking and keyword filtering

The extension includes a lightweight object tracking module that maintains object consistency between frames, shown in Figure 5.6. Objects are matched across frames based on their class and spatial proximity. This reduces visual flickering and improves perceived stability.

The object tracking system begins with ID generation and preservation. Each new detection is either assigned to a new identifier or matched with a previously seen object. Matching is determined by class ID similarity and spatial proximity using a position threshold. These matches are stored in an array of `previousDetections` and evaluated in the

`findMatchingDetection` function. Once an ID is assigned, it persists across frames, ensuring that the same object maintains its label and bounding box without blinking. This results in a stable visual overlay.

In addition to tracking, the system includes keyword-based filtering. The keyword entered by the user is retrieved from Chrome's local storage and used to compare against each detected object's class name. Matching is case-insensitive and, if successful, triggers a visual highlight. This includes changing the bounding box color to green and incrementing a match counter.

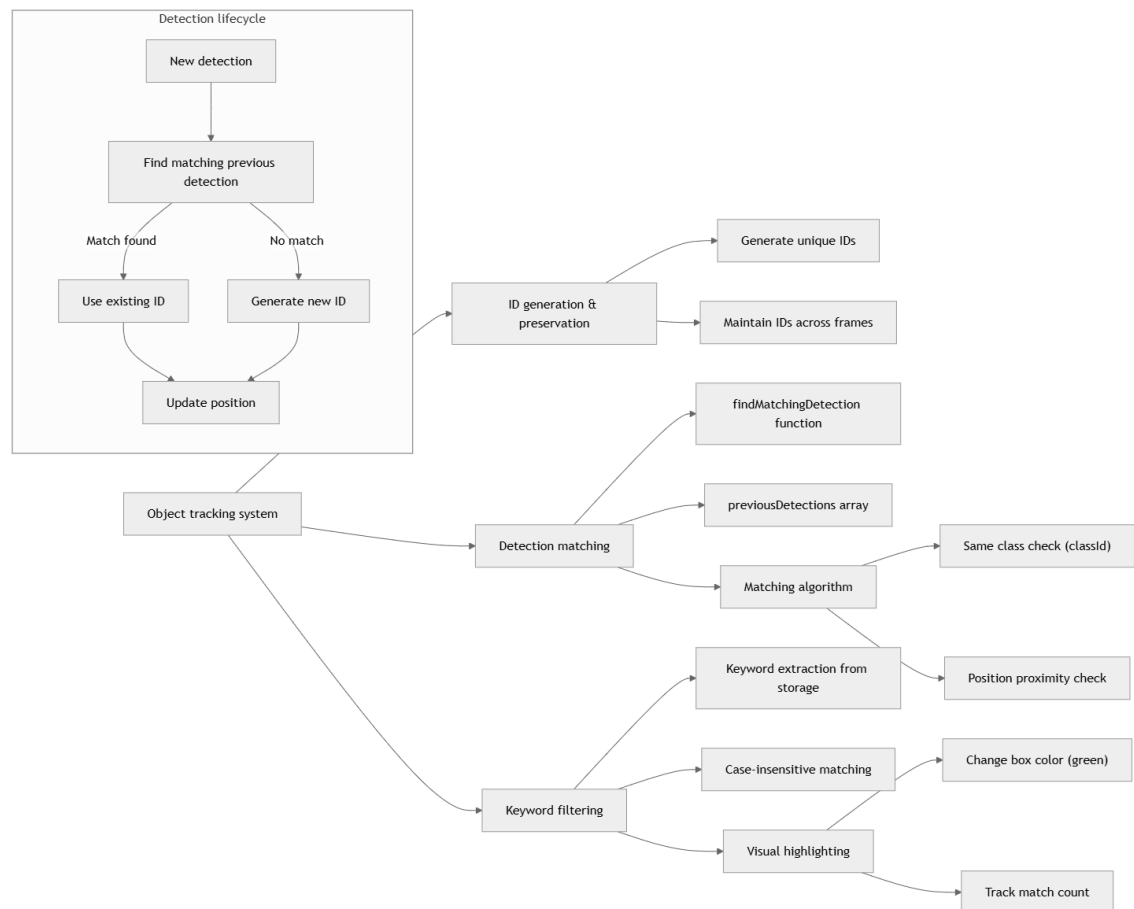


Figure 5.6. The lifecycle of object tracking and keyword filtering

5.7 Resource and performance management

Browser extensions, particularly those involving real-time computer vision models, can introduce significant computational overhead. Prior studies [17] demonstrate that even non-malicious extensions can adversely affect browser responsiveness, energy efficiency, and page rendering performance.

The extension implements multiple strategies to manage memory and improve runtime performance. Performance is managed through a combination of adaptive throttling,

memory and visibility management, backend acceleration, and DOM optimization, as shown in Figure 5.7.

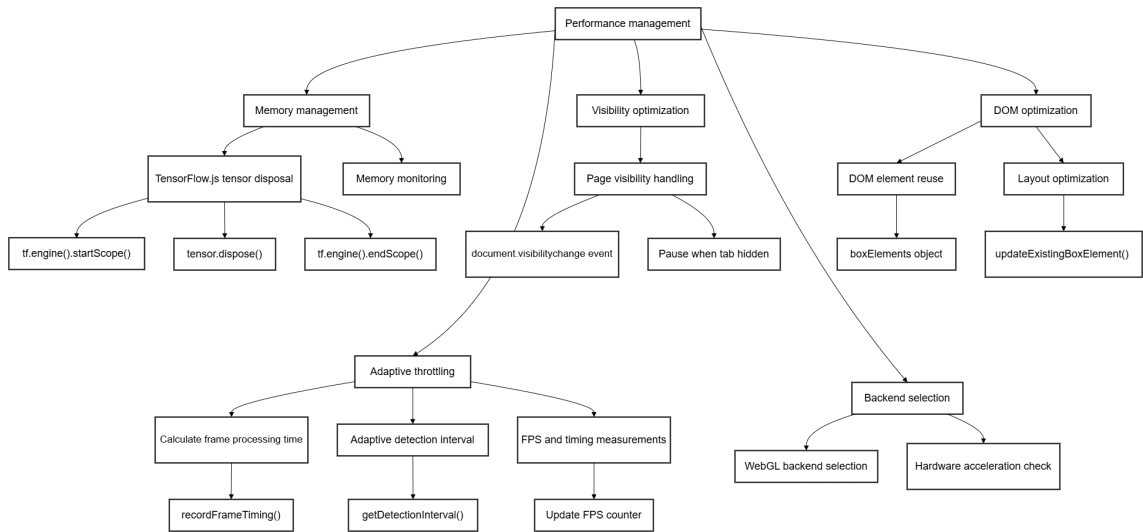


Figure 5.7. Performance optimization

The adaptive throttling mechanism dynamically adjusts the interval between detection frames based on real-time measurements of processing latency. Each frame's processing time is logged via `recordFrameTiming()` and used by `getDetectionInterval()` to calculate an appropriate interval for the next iteration. This allows the system to maintain a balance between performance and resource load.

Memory management is implemented using TensorFlow.js's built-in memory cleanup functions. By calling `tf.engine().startScope()` and `tf.engine().endScope()` around the inference logic, temporary tensors are automatically tracked and disposed of after use. Explicit disposal via `tensor.dispose()` is also used to eliminate potential memory leaks during prolonged detection sessions.

Visibility optimization is handled using the `document.visibilitychange` event listener. When the browser tab becomes hidden, the system automatically pauses detection to conserve computational resources and resumes when the tab becomes active again.

DOM optimization is achieved through the reuse of bounding box elements stored in the `boxElements` object. Rather than regenerating new elements on each frame, existing DOM nodes are updated in place using `updateExistingBoxElement()`. This approach minimizes layout recalculations and leads to smoother visual updates.

Finally, backend selection ensures the best available computational resources are used. The WebGL backend is selected for TensorFlow.js operations to take advantage of GPU acceleration where available. In environments lacking GPU support, fallback mechanisms maintain compatibility using CPU-based computation.

6. RESULTS AND DISCUSSION

6.1 Performance results

The Chrome extension successfully achieves real-time object detection entirely within the browser using TensorFlow.js and the YOLO11n model. The system detects 80 COCO classes, highlights keyword-matched objects, and delivers interactive visual feedback without transmitting any user data externally. It maintains modular design and stable performance across supported devices.

To evaluate the real-time performance of the object detection Chrome extension, it was tested on two different laptop configurations representing average and high-performance environments. The average FPS is displayed at the bottom of the detection interface.

The method used for calculating FPS was embedded directly into the detection loop of the Chrome extension. This implementation relies on high-resolution timestamps provided by `performance.now()` and a per-frame counter (`modelInferenceCount`) to determine how many frames were processed by the YOLO model over each one-second interval. An array (`processingTimes`) was also used to compute the average inference time per frame. These values were periodically updated in the user interface to reflect current system performance. This method provided consistent insight into the model's raw inference throughput and was used across both development and test machines for direct comparison.

The first laptop, used throughout the development process, was a Dell Latitude 3520 equipped with an 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, featuring 4 physical cores and 8 logical threads, and 20 GB of RAM, running on Windows 11. This laptop lacks a discrete GPU and relies on integrated graphics with TensorFlow.js WebGL backend. The extension consistently achieved an average inference rate of 7 FPS during screen-sharing detection sessions.

In contrast, a high-performance ASUS TUF Gaming F15 FX506HCB was used for testing on a more capable system. This device features a 2.7GHz Intel Core i5-11400H processor, 16 GB RAM, and a dedicated NVIDIA GeForce RTX 3050 GPU running Windows 11. On this system, the extension achieved an average inference speed of 28 FPS, thanks to improved GPU acceleration, higher memory bandwidth, and faster processing of DOM

updates and model inference.

In addition to the application's internal logging, Chrome's built-in Developer Tools were used to validate performance on the Dell Latitude 3520 system. A 10-second screen recording session was captured under the "Performance" tab, allowing detailed analysis of frame intervals and rendering behavior, as shown in Table 6.1. In this case, the frame duration for the extension was observed to fluctuate but averaged around 133 ms per frame, indicating approximately 7 FPS. In comparison, Hyuto's YOLOv8-TFJS web application demonstrated an average frame duration of 116 ms, equivalent to roughly 9 FPS, when tested under identical conditions. These numbers reflect both inference and UI rendering performance, as perceived by the browser's rendering engine.

Table 6.1. Performance measurements of two object detection applications

Application	YOLO-based Object Detection Chrome Extension	Hyuto's YOLOv8-TFJS Web Application [12]
Total time (seconds)	10.28	10.33
Scripting (ms)	5163	3597
System (ms)	258	268
Rendering & Painting (ms)	123	17
FPS	7	9

These measurements indicate that while both applications maintain comparable total execution times (10.3 seconds), the Chrome extension has a higher scripting cost (5163 ms vs. 3597 ms). This suggests the extension performs more complex or frequent JavaScript operations per frame, possibly due to DOM-based bounding box manipulation and keyword-based filtering. On the other hand, the web-based app achieves lower rendering overhead, which may be attributed to its use of canvas rendering rather than DOM overlays. These insights are useful for identifying areas of potential improvement, such as optimizing the update logic for detection overlays and offloading more rendering operations to the GPU using canvas or WebGL directly.

It must be acknowledged [1] that browser-based machine learning inference will always face inherent resource limitations compared to native applications. Future improvements may involve more aggressive frame prioritization, offloading rendering tasks to GPU-accelerated canvas elements, and exploring lightweight model variants to further reduce computational costs.

6.2 Security and privacy challenges and solutions

Deploying deep learning models in browser environments as Chrome extensions offers significant advantages in terms of accessibility and user experience. However, it simultaneously raises important concerns related to security and privacy. Careful design and implementation choices are necessary to ensure that these factors are addressed adequately.

Chrome extensions operate within a security framework that separates privilege domains to mitigate risks [18]. Chrome employs mechanisms such as privilege separation between content scripts and background scripts, isolated execution environments known as "isolated worlds" and a strict permission system where extensions must declare the APIs and host permissions they intend to use.

Despite these protections, vulnerabilities may still arise if developers mishandle extension privileges, expose sensitive data, or fail to sanitize dynamic content properly. In this project, security best practices were followed. The extension only requests minimal permissions, specifically, `activeTab` and `storage`, and loads all resources over secure HTTPS connections. DOM manipulations for visualizing detections are conducted carefully, avoiding dynamic script execution methods such as `eval`. Furthermore, no external third-party content is injected at runtime, and the extension design follows the principle of least privilege to minimize risk exposure. However, security in browser extensions is an evolving challenge. Future work could include enforcing a strict Content Security Policy (CSP), adopting even stricter permission models, and further isolating potentially sensitive operations to reduce exposure to cross-site vulnerabilities.

A key motivation for the design of this Chrome Extension was the preservation of user privacy. Unlike server-side machine learning systems, where user data must be transmitted externally for processing, the entire object detection pipeline in this project operates locally within the user's browser environment. This architecture ensures that screen content, detection results, and user behavior data remain strictly confined to the user's device.

The privacy advantages of this approach are significant. Users do not need to trust external servers with their private screen content, and no personal data is transmitted over the network. However, even fully client-side extensions must be designed carefully to avoid unintentional data leaks through exposed extension APIs or insecure cross-origin messaging [18].

6.3 Limitations and potential improvements

During the development process, several limitations emerged that reveal opportunities for future improvement.

One issue was the visual instability of detection boxes, which occasionally flickered or blinked during runtime. This behavior was largely due to the timing mismatch between DOM updates and frame rendering, particularly when detection results changed rapidly between frames. Although object tracking logic was implemented to help preserve box positions across frames, the lightweight tracking system sometimes failed to maintain consistent identities, especially under fast-moving or complex scenes. This inconsistency can negatively affect the user experience and reduce the perceived accuracy of the system. More advanced tracking algorithms and a switch from DOM-based overlays to canvas-based rendering may further enhance performance and visual smoothness by reducing layout recalculations and rendering overhead.

Performance also varied depending on screen content and available hardware. On lower-end devices or under heavier system load, a slight detection delay was observed between the appearance of an object and the rendering of its bounding box. This delay is inherent in the asynchronous nature of inference and post-processing in TensorFlow.js, and it becomes more pronounced when resource availability is limited. To improve this, future versions could experiment with refined frame throttling mechanisms.

From a build and deployment perspective, the project faced limitations with JavaScript bundling. The Vite build process occasionally produced warning messages related to chunk sizes exceeding 500 kB, primarily due to the inclusion of TensorFlow.js dependencies. Although this did not break functionality, it could negatively impact initial loading performance, particularly for users with slower network connections. While the chunk size warning can be suppressed through configuration, a more sustainable optimization would involve refactoring the project to use dynamic imports or manual chunk splitting to better distribute code bundles.

Lastly, future development could explore additional features such as push notifications triggered when specific objects are detected, or model update and variety. This would expand the utility of the extension beyond real-time visualization, particularly in safety-critical or surveillance contexts where users may want to be alerted without actively watching the screen.

7. CONCLUSION

This thesis presented the development and evaluation of a real-time object detection system implemented as a Chrome extension using TensorFlow.js and the YOLO11n model. The project successfully demonstrated that it is technically feasible to deploy deep learning-based object detection directly in the browser, enabling privacy-preserving and client-side inference without the need for backend servers.

The extension offers a modular and maintainable architecture, with distinct layers for user interaction, detection logic, model management, and browser integration. Through effective use of WebGL acceleration and adaptive performance strategies, the system maintained reliable operation under typical screen-sharing conditions. On the development device, inference performance reached 7 frames per second, demonstrating that real-time detection is achievable on modest hardware.

While the implementation fulfilled its primary objectives, several challenges emerged during the process. These included compatibility issues during model export on Windows, occasional rendering artifacts such as blinking detection boxes, and varying inference latency. Nonetheless, these limitations provided valuable insights into the constraints of browser-based machine learning and informed suggestions for future improvements.

Opportunities for future enhancement include introducing support for push notifications, improving object tracking algorithms, and expanding model flexibility. The emergence of alternative inference frameworks such as ONNX Runtime for JavaScript [19] and JS-Pytorch [20] further emphasizes the evolving landscape of web-based AI applications.

In conclusion, this project contributes a practical and extensible example of how advanced machine learning models can be embedded into everyday browser workflows. It highlights the potential of Chrome extensions as a platform for deploying interactive, AI-powered tools, and lays the groundwork for future research and development in this domain.

REFERENCES

- [1] Rivera, J. D. D. S. *Practical TensorFlow.js. Deep Learning in Web App Development*. Apress Berkeley, CA, 2020. 303 p. DOI: <https://doi.org/10.1007/978-1-4842-6273-3>.
- [2] Smilkov, D., Thorat, N., Assogba, Y., Yuan, A., Kreeger, N., Yu, P., Zhang, K., Cai, S., Nielsen, E., Soergel, D., Bileschi, S., Terry, M., Nicholson, C., Gupta, S. N., Sirajuddin, S., Sculley, D., Monga, R., Corrado, G., Viégas, F. B. and Wattenberg, M. *TensorFlow.js: Machine Learning for the Web and Beyond*. 2019. arXiv: 1901.05350 [cs.LG]. URL: <https://arxiv.org/abs/1901.05350>.
- [3] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.
- [4] *Deploy YOLO with TensorFlow.js*. Ultralytics. URL: <https://docs.ultralytics.com/integrations/tfjs/> (visited on 12/27/2024).
- [5] *Chrome Extensions | Chrome for Developers*. URL: <https://developer.chrome.com/docs/extensions> (visited on 12/15/2024).
- [6] Frisbie, M. *Building Browser Extensions. Create Modern Extensions for Chrome, Safari, Firefox, and Edge*. Apress Berkeley, CA, 2023. 538 p. DOI: <https://doi.org/10.1007/978-1-4842-8725-5>.
- [7] Goh, H.-A., Ho, C.-K. and Abas, F. S. Front-end deep learning web apps development and deployment: a review. *Applied Intelligence* 53 (2023), pp. 15923–15945. URL: <https://doi.org/10.1007/s10489-022-04278-6>.
- [8] *Developer Survey*. StackOverflow. 2024. URL: <https://survey.stackoverflow.co/2024/technology> (visited on 04/28/2025).
- [9] Cass, S. *The Top Programming Languages 2024*. IEEE Spectrum. Aug. 22, 2024. URL: <https://spectrum.ieee.org/patent-power-2025> (visited on 04/28/2025).
- [10] Khanam, R. and Hussain, M. *YOLOv11: An Overview of the Key Architectural Enhancements*. 2024. arXiv: 2410.17725 [cs.CV]. URL: <https://arxiv.org/abs/2410.17725>.
- [11] *TensorFlow.js Deployment Example : Browser Extension*. Tensorflow. URL: <https://github.com/tensorflow/tfjs-examples/tree/master/chrome-extension> (visited on 12/15/2024).
- [12] Hyuto. *Object Detection using YOLOv8 and Tensorflow.js*. 2023. URL: <https://github.com/Hyuto/yolov8-tfjs> (visited on 12/15/2024).

- [13] Burlacu, D. *Video Object Detection. Chrome Extension*. Version 1.1. 2024. URL: <https://chromewebstore.google.com/detail/video-object-detection/igoognpcnlilhlpjmedmicfggehaegck> (visited on 12/15/2024).
- [14] *Ultralytics YOLO11*. Ultralytics. URL: <https://docs.ultralytics.com/models/yolo11/> (visited on 02/21/2025).
- [15] *Known Issues | TensorFlow Decision Forests*. TensorFlow. URL: https://www.tensorflow.org/decision_forests/known_issues (visited on 02/23/2024).
- [16] Terven, J., Córdova-Esparza, D.-M. and Romero-González, J.-A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Machine Learning and Knowledge Extraction* 5.4 (Nov. 2023), pp. 1680–1716. ISSN: 2504-4990. DOI: 10.3390/make5040083.
- [17] Jin, B., Li, H. and Zou, Y. Impact of extensions on browser performance: An empirical study on google chrome. *Empirical Software Engineering* 30 (Apr. 2025). DOI: 10.1007/s10664-025-10633-1.
- [18] Nicholas Carlini, A. P. F. and Wagner, D. An evaluation of the Google Chrome extension security architecture. *21st USENIX Security Symposium* (2012). URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final177_0.pdf.
- [19] *Get started with ORT for JavaScript*. ONNX Runtime. URL: <https://onnxruntime.ai/docs/get-started/with-javascript/> (visited on 03/11/2025).
- [20] *PyTorch in JavaScript*. Eduardo Leitão da Cunha Opice Leão. URL: <https://github.com/eduardoleao052/js-pytorch> (visited on 03/11/2025).