POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Image Classification in the Browser: a performance assessment

Supervisors

Prof. ANDREA CALIMERA

Dott. VALENTINO PELUSO

Candidate

VALERIA SORRENTI

April 2023

Summary

During the last decades, are made steps forward in the Artificial Intelligence (AI) field. Until recently, the Cloud Computing paradigm has allowed for increasingly complex and large models, but recently, a paradigm shift has occurred, and Edge Computing has taken over, having an eye on issues such as privacy and portability. In the last years, JavaScript (JS) libraries have emerged, allowing Deep Learning (DL) to be brought into the browser. These libraries provide several benefits. In particular, they ensure portability. WebAssembly is a low-level binary format that is designed to be executed by web browsers. It provides a way to run code in the browser, in a way that is closer to native machine code than JS. It means that DL models built using JavaScript libraries can be deployed with WebAssembly on a wide range of devices and platforms, making it easier to integrate DL into web applications. The solution presented is a static web application that performs a classification task on the emotional states of people in a work environment. Using a static site ensures privacy, and the ONNX Runtime Web, enables ONNX DL models to run in the browser. The results are obtained by testing the web application of three devices with different hardware performances. Eight Convolutional Neural Networks with different depths and complexity are taken into consideration. The outcomes produced are the latency time and the prediction with its probability. This Master thesis, proves that JS libraries are the correct solution to overcome the issue of portability and, in particular, put the focus on the performance obtained with a web application.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

DNN

Deep Neural Network

CNN

Convolutional Neural Network

DNN

Recurrent Neural Network

DL

Deep Learning

ML

Machine Learning

JS

JavaScript

CPU

Central Process Unit

GPU

Graphic Process Unit

ONNX

Open Neural Network Exchange

TP
True Positive

FP
False Negative

ROC
Receiver Operating Characteristic

AUC
Area Under the Curve

IoT
Internet of Things

HTTP
HyperText Transfer Protocol

URL
Uniform Resource Locator

HTML
Hyper Text Murkup Language

GIF
Graphics Interchange Format

TCP/IP
Transmission Control Protocol Over Internet Protocol

SQL
Structured Query Language

DOM
Document Object Model

CSS
Cascading Style Sheets

SSH	Secure Socket Shell
DSLR	Digital Single-Lens Reflex
FLOPs	loating-Point Operation per second
NAS	Neural Architecture Search
SGD	Neural Architecture Search
VM	Virtual Machine
ES	ECMAScript
NLP	Natural Language Procssing

Chapter 1

Introduction

In the last decades, Artificial Intelligence (AI) has been widely used to automate industrial processes in different sectors and to make easier some actions in everyday life. Many applications concern image processing, speech recognition, object training, and others. They combine a large amount of **data** and algorithms for solving assigned tasks. In several context, technologies that exploit AI need **sensitive data**, these concern:

- personal data that reveal political orientation, ethnic origins, philosophical beliefs
- genetic data, biometric data that are used exclusively to identify a human being
- data related to the health conditions
- data about sex or sexual orientation
- trade-union membership

In recent years, Computer Vision has been widely exploited for implementing numerous applications. In this field, data processed and passed to DL models are images. If the subject of an image is a person or object that contains data traceable to a person, it means that this data contain sensitive data.
For example, if a company would make statistics into the working environment or the collective productivity, it would need sensitive data of a worker, in his office or smart-working setting, for making predictions. In the majority of cases, these data are sent to third parties, and people do not accept that they can get hold of these data, to exploit them for other scopes. The big problem that emerges is the **privacy** of users when sending sensitive data to third parties to exploit AI technologies.

To overcome this problem, AI community studies methods to bring the inference phase near as possible to the data source. A change of paradigm is necessary. The paradigm widely used in AI technologies is Cloud Computing: data are sent and processed on cloud servers. This paradigm, especially for Deep Learning (DL) applications, is widely adopted because for processing data and for using DL models, it is necessary hardware that has optimal performances, sometimes suffering long response time.
Unfortunately, this paradigm does not take care of users' data privacy. Another paradigm that, instead, takes care of this aspect is edge computing [1]. It brings the computation in the proximity of the data source preserving the users' privacy, contrary to cloud computing. The resulting limitations of edge computing are in the hardware performances because not all devices have powerful Graphic Process Units (GPUs) and Central Process Units (CPUs) able to process heavy data and use heavy DL models.

Edge computing solves privacy issue but brings attention to another problem. Due to the heterogeneity of operating systems and hardware, an AI application should have one version for each operating system and hardware. Developing and keeping updated every single version of an application is a non-trivial task. Native applications exploit DL frameworks and libraries that could run on heterogeneous development environments as Windows, Linux, MacOS/iOS or Android. These applications, are implemented with different languages according to the development environments. The development of mobile applications, is harder than other devices. Often, applications must be maintain and develop in both iOS and Android versions. In addition, also distribution is non-trivial because the most current platforms have an app store, and sometimes, apps are rejected or need manual test before being accepted. The solution is to develop **portable** applications over operating systems and hardware devices.

A DL application that ensures **privacy** and **portability** issues is a **web application** that makes **inference on browser**. For building AI web application many **JavaScript** (JS) libraries are developed. They provide the instruments and APIs required for allowing DL models to be integrated in the web application that make accessible **CPU** and **GPU** to DL models.
Another technology that runs together with JS components is **WebAssembly** [2]. It is a low-level binary format that is designed to be executed by web browsers. It provides a way to run code in the browser that is closer to native machine code than JS. One of the main use cases for WebAssembly is to allow developers to run computationally intensive code in the browser, such as ML algorithms. Using WebAssembly, developers can create web applications that can take advantage of the full power of the users' devices without relying on a server to perform

complex computations. Most surveys are made for comparing the performance between native and in-browser applications [3]. From the above mentioned studies, it is clear that the performances of native applications are more powerful in terms of inference time, but the backend technologies (CPU or GPU) have a strong impact.

In this work, a web application is realized, able to recognize the emotional states of a user through a photo of him/her face. For its relevant and flexible features, **Open Neural Network Exchange** (ONNX) model format [4] and **ONNX Runtime Web** [5] are chosen for developing a web application that performs inference on an image classification task. It ensures data privacy and solves the problem of portability among different devices and operating systems. ONNX is chosen for its speed and its **interoperability**. The last property is very advantageous because there are no limits for frameworks used to implement a model and for the runtime used to deploy the model. The Web Application developed has the task of predicting the emotional state of a worker through a picture taken by a webcam of a notebook or other devices. The work is organized into three main parts:

- training Pytorch [6] models
- conversion of it in ONNX format
- integration of ONNX model in a Web Application

In order to satisfy portability on different devices with different computational performances, many DL models are trained because not all devices are the same hardware performances. The web application developed give back the **prediction** produced by the inference and the **latency** time for making prediction.
The setting adopted for discussing this work is the following. Chapter 2 discusses about some background concepts, useful for understanding the operations and behind the application developed, and it will mention some applications developed. Chapter 3 describes the flow followed for the development of the application. Chapter 4 explain the experiments and will comment on the results obtained. Finally, Chapter 5 gives comments on the goodness of the application, makes comparison between ONNX and Pytorch models and gives some ideas for future works.

Chapter 2

State of Art

This chapter introduces the fundamental notions for understanding the operations and motivations behind using browsers for ML inference. The first part deals the theoretical **background**, and in the second part the **applications** developed in recent years.

2.1 Background

This section aims to give some fundamentals of **DL**, **edge computing**, and **DL inference in browser** in such a way as to link each component for a better understanding of the final solution.

2.1.1 Deep Learning

DL is a subset of AI, particularly Machine Learning (ML). This field is based on particular algorithms, **Artificial Neural Network** (ANN) with multiple layers, used to analyze and understand complex data, such as images, audio, and text. DL is present many applications, including image recognition, speech identification, natural language processing, and self-driving cars.

2.1.1.1 Artificial Neural Networks

ANN is an algorithm of DL that tries to imitate the human brain and its behaviour. An ANN is made of multiple layers of nonlinear processing unit (neurons). They are used to transform data and extract information from them. Each output of a single layer is the input of the following one, until the final prediction is made. The three essential blocks that make up an ANN are **input layer**, **hidden layers** and **output layer**.

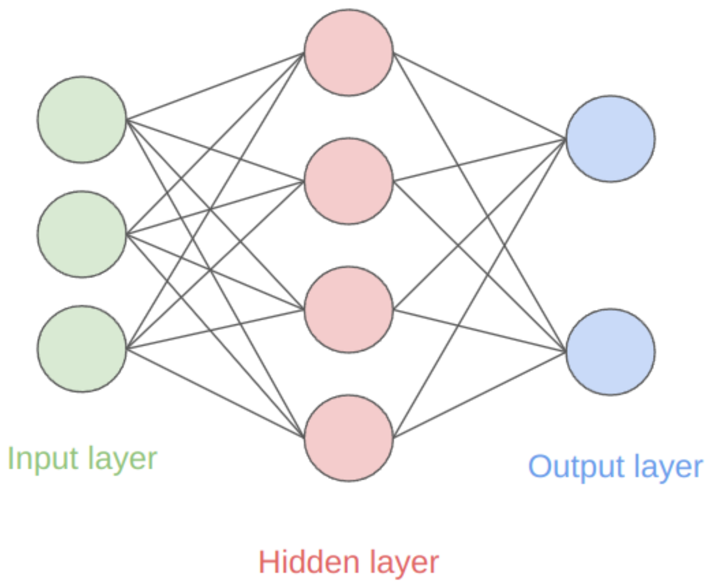


Figure 2.1: Example of an Artificial Neural Network with one hidden layer.

Figure 2.1 shows a simple scheme of a ANN. The idea of ANN has its roots in the years 1940s and 1950s, when developers explored the possibility of creating computer systems that could mimic the structure and function of the human brain for the first time. Warren McCulloch and Walter Pitts published a seminal paper in 1943 that proposed using simple mathematical models to simulate the behavior of individual neurons [7], and this work laid the foundation for the development of the first ANNs. In the 1950s, Frank Rosenblatt developed the Perceptron [8]. It is the simplest neural network that have n number of inputs, one neuron and one output.
However, the Perceptron was limited in its ability to solve more complex problems, and it was later found that single-layer perceptrons could only learn linearly separable functions. In the 1960s and 1970s, the researchers developed multi-layer perceptrons (MLPs) and other types of ANNs that were capable of solving more complex problems by stacking multiple layers of artificial neurons.

The output z (eq. 2.2) of each neuron is obtained through many computations. The first operation is the **dot product** between **input** x and **weights** w (the strength of connections) (eq. 2.1). Then, all dot products are summed.

$$x \cdot w = (x_1 \times w_1) + ... + (x_n \times w_n) \tag{2.1}$$

Then a **bias** b (eq. 2.2) is added to the function produced above (eq. 2.1)0.

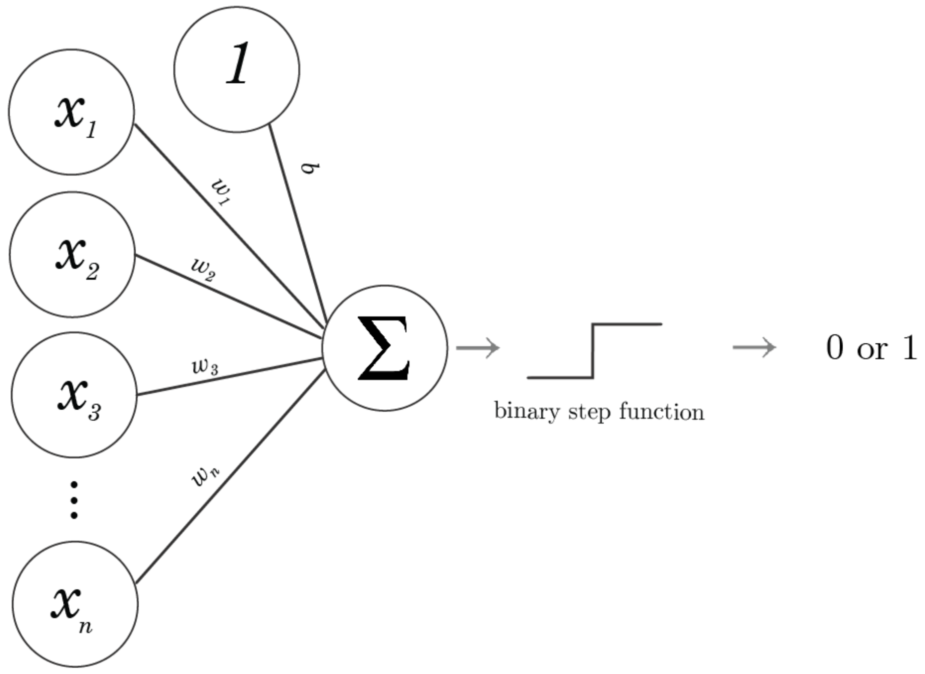


Figure 2.2: Architecture of Perceptron. Source [9]

$$z = x \cdot w + b \tag{2.2}$$

Finally, the output z is passed to a **non-linear activation function** σ to make the function bounded. For the example of sigmoid for non-linear activation function, the output obtained is $\hat{y}$ (eq. 2.3).

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

This flow of operation is called **forward propagation**, in which is computed the output z. Among ANN there are three basic structures:

- **Deep Neural Networks** (DNN): feedforward networks with multiple layers that bring the output forward without going back.
- **Convolutional Neural Networks** (CNN): variation of multi-layer perceptron used for tasks that require minimal preprocessing, especially in images.
- **Recurrent Neural Networks** (RNN): networks with connections between nodes that create a direct graph, used to store time sequences.

Different fields use these algorithms and their combinations, such as computer vision, language, and audio recognition, bioinformatics.

2.1.1.2 Training

The learning process of an ANN is called training. In a few words, the DL model learns how to make accurate predictions, computing the proper weights and biases. A model needs to **train data** to learn the task and the structure of objects he has to process. This data could be labeled or unlabeled. The first case is present in **Classification** or **Regression** tasks data also contain the **class** to which it belongs (the label is called *ground truth*). Instead, the second case is present in **Clustering** tasks data contain only itself. There are two macro-approaches of learning: **Supervised** and **Unsupervised**. For Unsupervised Learning task during the training phase, the network receives several unlabeled data. The aim of this process is to extract information from data for learning as well as possible because there aren't labels to compare.
For Supervised Learning, during the training phase, the network receives many labeled data. Here, the goal is to minimize the error computed between the prediction made by the model and the *ground truth.*
Only Supervised Learning will be described in depth because it is necessary to know its fundamental to understand the next sections.
Another fundamental parameter of the training process is **Loss function**. The loss function is a function cost and it is used to compare the prediction and the *ground truth.* It measures how well a network performs. Loss functions can be divided into two basic groups: **Regression Loss** functions and **Classification Loss** functions. The output's model, used for the Regression task, is a real value, while for the Classification task is a probability distribution. The **softmax layer** to the end of the network is used to convert digits into probabilities.
This thesis work uses Classification for its task. Among loss functions, the widely used for this task is the Cross-Entropy Loss L (eq. 2.4). It is computed as the negative sum of all n products between the prediction y_i and the natural logarithm of correspondent *ground truth* $\hat{y}_i$. It measures the distance between the prediction y and the *ground truth* $\hat{y}$.

$$L(y, \hat{y}) = -\sum_{i=1}^{n} y_i * \ln(\hat{y}_i) \tag{2.4}$$

The training process has two fundamental phases:

- **Backpropagation**: the process aims to minimize the Loss function L. For doing it, is necessary to tune the weights and biases. This values can be obtained through the computation of the negative gradient of Loss function δL

with respect to the weights w. The gradient is the vector of partial derivatives of a function f in a point x, it is used to find a local minimum of a multi-variable function. Since the loss function is not directly related to the weight w_i, it necessary use the chain rule (eq. 2.5), where L is the loss function, z is the output value of the dot product summation with addition of bias b in (eq. 2.2) and $\hat{y}$ is the output of the activation function in (eq. 2.3).

$$\frac{\delta L}{\delta w_i} = \frac{\delta L}{\delta \hat{y}} \times \frac{\delta \hat{y}}{\delta z} \times \frac{\delta z}{\delta w_i} \tag{2.5}$$

- **Optimization**: the process aims to find the best weights and biases among the possible solutions. Optimization algorithm makes this operation. To explain the concept it is chosen Gradient Descent, that changes the weight and biases values proportional to the negative Loss function with respect to the corresponding weight and bias. Weight and biases are initialized randomly, but during iteration are updated as in (eq. 2.6) and (eq. 2.7). The factor α is called **learning rate**, it is an hyperparameter that controls how change weight and bias.

$$w_i = w_i - (\alpha \times \frac{\delta L}{\delta w_i}) \tag{2.6}$$

$$b = b - (\alpha \times \frac{\delta L}{\delta b}) \tag{2.7}$$

TThe procedure described above, is repeated until convergence. Other important hyperparameters in training phase are the **epochs**, i.e., the iterations in which data passes exactly one time in one of them, and the **batch size**, i.e., is the number of samples processed before the model update.

2.1.1.3 Dataset Handling and Data Preprocessing

The choice of the right data and its handling is fundamental. This part of the entire DL process is the first that affects the goodness of the final result.
During training, the DL model learns the task assigned through data. If data processed in the training phase are wrong or have the incorrect quantity, it is possible to incur two bad situations:

- **Overfitting**: The condition in which the model becomes too complex, fits the training data too well and it becomes not able to generalize to examples not present in training data. This occurs when a model is trained on a limited amount of data and it starts to learn the noise in the data rather than the underlying patterns.

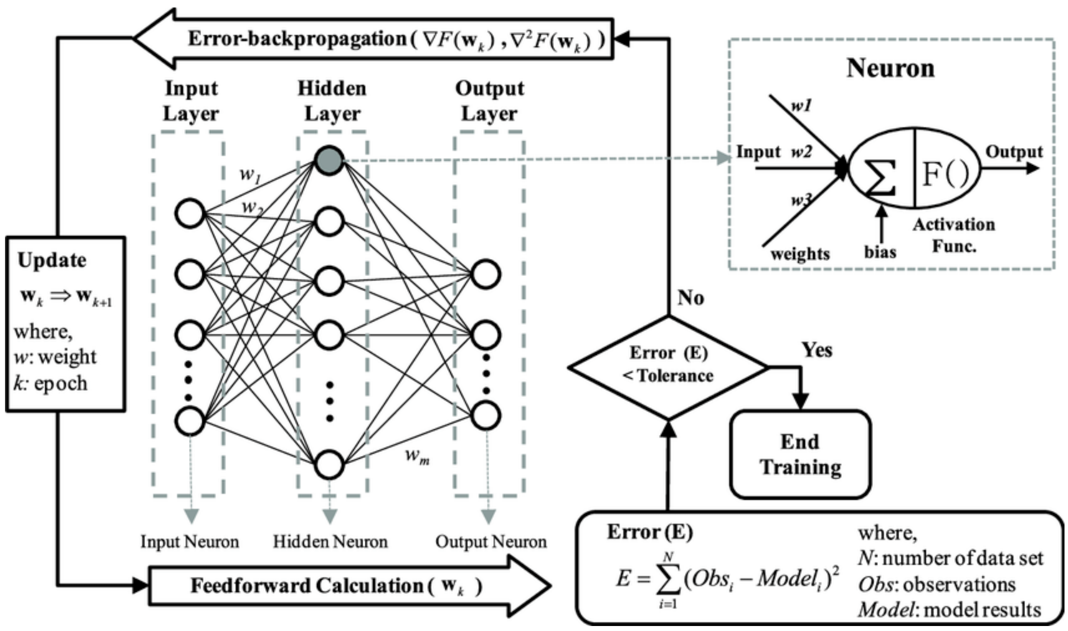


Figure 2.3: Scheme of training algorithm. Source [10]

- **Underfitting**: the condition in which a model is too simple, is not able to capture the underlying patterns in the training data and it is not able to generalize to new data.

In order to train and test the model, dataset is split in subset:

- **Training dataset**: data used to train model. Usually it is the most significant slice of the source dataset.
- **Validation dataset**: data used to evaluate the model during the training phase.
- **Test dataset**: data never seen by the model, used to test it at the end of training phase.

Among different splitting techniques, the most widely used for a large dataset is the **Random split**. The source dataset is shuffled and samples are picked randomly and put in one of the split dataset, in a proportion choose by the user.
There are other famous techniques as **Stratified**, similar to Random but used when datasets have imbalanced class distribution, and **K-Fold Cross Validation**, the model is train and evaluated "K" times on different samples. The last cited technique is the most robust, but it is not recommended for large dataset.
After the split phase, usually, data are pre-processed and normalized in order

to have comparable output. If data are unbalanced in class distribution, there are over-sampling or under-sampling techniques to fit the model with the correct proportion of classes. Data augmentation techniques are useful in the training process, in order to avoid overfitting.

2.1.1.4 Evaluation Metrics

For classification tasks, there are different evaluation metrics. Their role is to test the goodness of a model, in other words, if a model can make correct predictions. For simplicity, we consider a binary classification task whose result can be only *Positive* or *Negative.* This prediction could be *True Positive (TP)*, *False Positive (FP)*, *True Negative (TN)* and *False Negative (FN)* with respect to the *ground truth.*
The metrics considered are:

- **Accuracy**:
  $$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.8}$$
  It is the commonly used metric. It computes the positive percentage over all cases. It gives no information on the distribution of *FP* and *FN* if classes are unbalanced, the combinations with other metrics are more suitable.

- **Precision**:
  $$Precision = \frac{TP}{TP + FP} \tag{2.9}$$
  Its value depend from both *Negative* and *Positive* samples. It consider when a sample is classified *Positive*, but doesn't take care of classifying correctly all positive samples. It is often used in conjunction with another metric called Recall (eq. 2.10).

- **Recall**:
  $$Recall = \frac{TP}{TP + FN} \tag{2.10}$$
  It measure the ability of the model to detect positive samples. It consider the fairness of the classification of all *Positive* samples, but it doesn't care if a *Negative* sample is classified as *Positive.* It is often used in conjunction with Precision (2.9), which together provide a more complete picture of a model's performance. The Recall is particularly important when the costs of false negatives are high.

- **F1-score**:
  $$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.11}$$

It is the weighted average of Precision (eq. 2.9) and Recall (eq. 2.10). This metric is used when we have an unbalanced class or when you care more about the false positives and false negatives than the overall accuracy. It is more robust than the previous ones.

- **ROC Curve and AUC**: The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performances of a binary classification model, and the Area Under the ROC Curve (AUC) is a scalar metric that summarizes the overall performance of the model.

$$FPR = \frac{FP}{FP + TN} \tag{2.12}$$

The ROC curve plots the so called 'true positive rate' (TPR) against the false positive rate (FPR) at different classification thresholds. The TPR is also known as Recall (eq. 2.10), and the FPR is the rate of false positives among the negative instances. The AUC is the area below the ROC curve and provides a single number that summarizes the model's performance, with a value of 1 indicating perfect performance and a value of 0.5 indicating a model that performs no better than random guessing.
AUC is useful metric when the data is imbalanced or when the costs of false positives and false negatives are different. A model with a higher AUC has better performance than a model with a lower AUC.

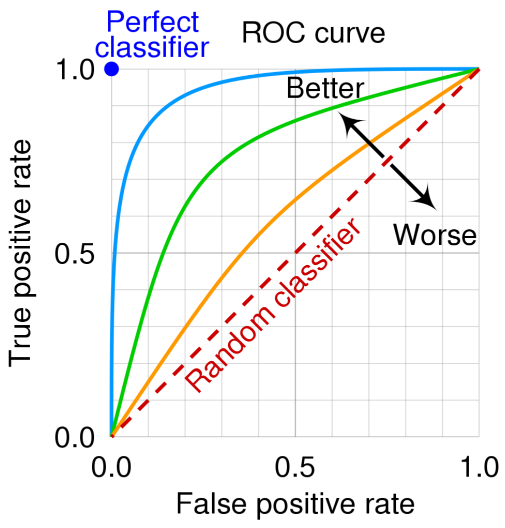


Figure 2.4: Example of ROC curve. The AUC is respectively the area under the ROC. In this figure is highlighted how a classifier is good or bad. Source [11]

2.1.1.5 Image Classification and Convolutional Neural Networks

Image classification is a field of computer vision. The goal is to recognize and categorize objects or scenes in images. The image classification goal is to assign an input image to one of a pre-determined set of labels or classes.
Models used for this task are **Convolutional Neural Network** (CNN). The architecture of CNN is arranged as the connectivity patterns of the human brain and is inspired by the visual cortex. A CNN can capture the spatial and temporal dependencies in an image. It reduces the number of parameters involved and reuses weights.

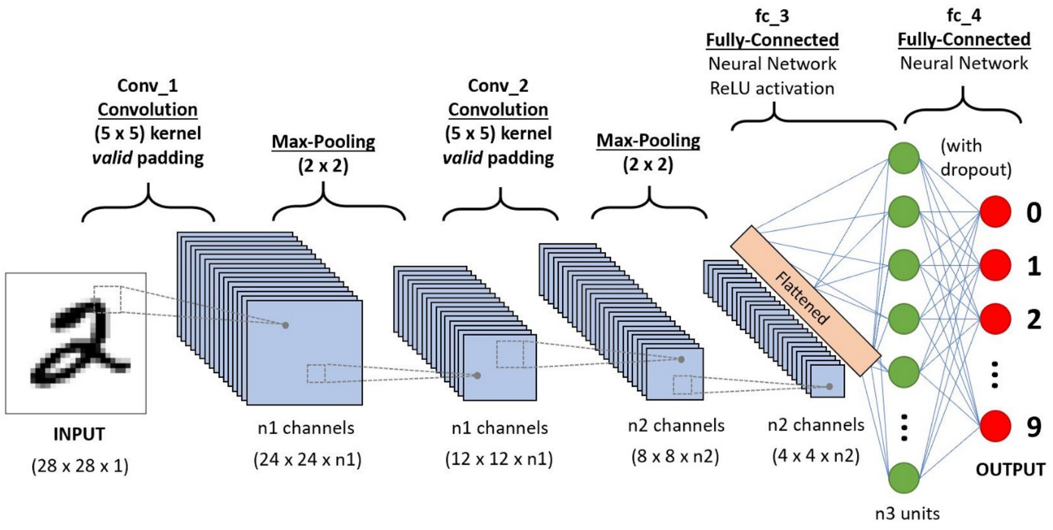


Figure 2.5: Example of convolutional neural network. Source [12]

Typically, in a CNN there are three layers: **convolutional** layer, **pooling** layer and a **fully connected** layer.
The **convolutional layer** is the heart of a CNN and it is used for feature extraction. The main idea behind convolutional layers is to apply set of filters to the input data to extract features, i.e., edges, textures, and shapes. This layer executes a dot product between two matrices, one is called kernel or filter (i.e, the set of learnable parameters) and the other one is a portion of the receptive field. During the forward pass, the kernel runs across the height and width of the image, producing a representation of the receptive region. This process produces a bi-dimensional representation of the image called the activation map, which provides the response of kernel in each spatial region of image. The sliding dimension of the kernel is called stride.
For example, if we have an input image of size $W \times W \times D$ and D_{out} number of

kernels with a spatial size of F, stride S and amount of padding P, the size of output volume is obtained through the (eq. 2.13). The resulting output volume will be $W_{out} \times W_{out} \times D_{out}$.

$$W_{out} = \frac{W - F + 2 * P}{S} + 1 \tag{2.13}$$

The ideas that bring computer vision researchers to adopt convolutional layers are:

- **Sparse interaction**: every output unit interacts with every input unit.
- **Parameters sharing**: for obtaining an output, the weights applied in a input are also used in other inputs.
- **Equivariant representation**: input and output change in the same way.

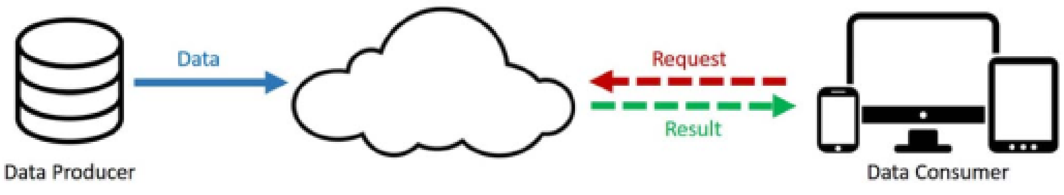
Convolution is a linear operation, but the images are not linear, so it is necessary to introduce **non-linearity layers** usually located directly after convolutional layers for introducing the non-linearity in the activation map. There are several types of non-linear operation, one widely used is the ReLU.
The **Pooling Layer** replaces the network's output at certain points deriving a statistic result from other nearby outputs. This operation helps to reduce the spatial dimension of the representation, decreasing the amount of computation and weights. Additionally, it is executed on every slice of the representation individually. There are different pooling operations, one of the most commonly used is max pooling, which retrieves the maximum output from the neighbourhood.
If the activation map of $W \times W \times D$, a pooling kernel with spatial size F and stride S the output volume, can be computed with (eq. 2.14). The output volume size will be $W_{out} \times W_{out} \times D$.

$$W_{out} = \frac{W - F}{S} + 1 \tag{2.14}$$

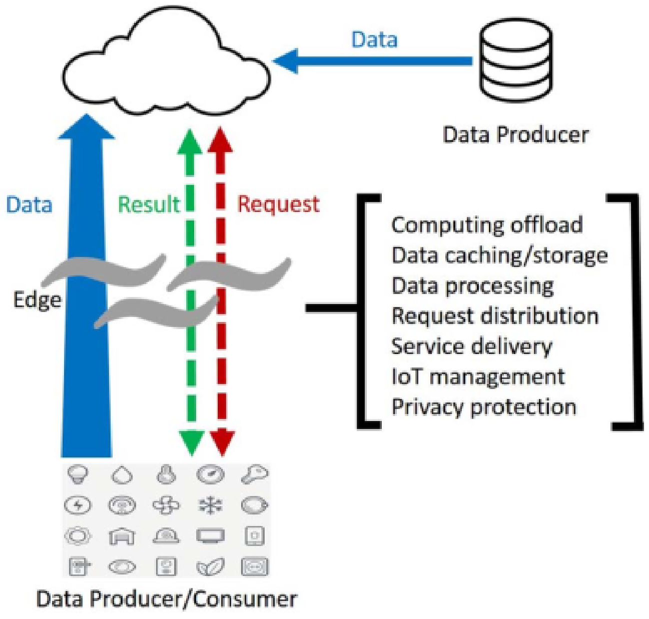
Fully Connected Layer contains all connections neurons wholly connected with all neurons of the previous and successive layers. For this reason, it is possible to compute a matrix multiplication with a bias effect. It is the last layer of a CNN architecture and is used to make predictions. The Fully Connected Layer helps to map the representation between input and output.
Another typical layer in a CNN is the **Dropout Layer**. It helps to prevent overfitting that, temporarily, doesn't allow some neurons to contribute, leaving the others active.

2.1.2 Edge Computing and paradigm shift

Edge computing is a species of distributed computing paradigm that brings computing power, storage, and other capabilities closer to the data source. The paradigm widely used before edge computing is **cloud computing**. Resources such as software and storage are provided over the internet rather than on a local computer or server. The remote servers host resources, or "the cloud," and users can access them from anywhere with an internet connection.
In the last years, a **shift paradigm** occurs.



(a) Cloud Computing paradigm.



(b) Edge computing paradigm.

Figure 2.6: Comparison between cloud and edge Computing. Figure 2.6a shows the cloud computing paradigm and is evident how data producer and consumer are two different figures. Figure 2.6b shows the edge computing paradigm and, unlike the previous paradigm, the data consumer is also the data producer. Source [1].

The factors that drive this shift are:

- **Privacy and security**: by processing data at the edge, there is less need to send sensitive data over a network, which can reduce the risk of data breaches.
- **Latency**: edge computing reduces the amount of time for processing data, because there is not necessity to send data to a centralized location first. This is important for applications such as autonomous vehicles, where decisions need to be made in real-time.
- **Bandwidth**: by processing data at the edge, fewer data needs to be sent over a network, which reduces the amount of bandwidth required.
- **Cost**: by reducing the amount of data that needs to be sent over a network, edge computing can also help to reduce costs associated with data transfer and storage.
- **Power consumption**: edge computing devices are typically smaller and less powerful than cloud computing servers, which can help to reduce energy consumption.
- **Reliability**: management of failures is crucial for the good working of a service. If a single node is out of service, users should be able to enjoy the service. This problem could be present also for less reliable connection technologies, hence in order to guarantee the service, each device must keep the same network topology of the entire distributed system.
- **Meeting the demands of Internet of Things (IoT) and 5G**: with the increasing number of IoT devices and the advent of 5G, there is a growing need for more localized computing capabilities to handle the large amount of data generated by these devices. Edge computing provides a solution to this problem by processing data at the network edge, close to the source of the data.

Nowadays, DL is widely used for AI applications, shifting towards a new paradigm: the Edge Intelligence [13]. DL needs high computational resources reach high results especially in the training phase. In order to satisfy the computational requirements of DL, the cloud computing paradigm is the obvious choice, exploiting the powerful resources of data centers. In order to use them, data needs to be moved from the sources to data centers, but this operation, in inference phase, raises some issues in terms of privacy. Due the discussed reasons above, Edge computing paradigm is a good solution for exploiting DL, enabling users **privacy**.

2.1.3 Web Application for AI

Edge computing is the perfect paradigm that enables the privacy of the users. By adopting this paradigm, devices process data directly on it. In this way, it is mandatory to develop different copies of the same application for each operating system or hardware performance. Thus, the edge computing paradigm leads to a **portability issue**. The solution is the **web browser**. In recent times, the AI community is attracted to experimenting with client-side DL. DL inference in a web browser enables real-time processing of data and applications on the client side without needing for a server or cloud. This can result in faster and more responsive experiences for the user, increasing the **privacy** and security of sensitive data, reducing server loads and costs, and solving the problem of **portability**. The notions of background necessary for understanding the components of a web application and how to exploit the browser to make DL inference will be explained in the rest of the section.

2.1.3.1 Web Application

A Web Application is an application program that uses browsers to run, thereby does not need to download. Two main components characterize the general architecture of a web application:

- **Client**: who sends requests to the server. In this case it is the web browser.
- **Server**: who holds information (e.g., Web sites) and responds to request information.

The network handles the communication between the client and the server. In the **server-side** there are different layers:

- **Web server** manages the Hyper Text Transfer Protocol (HTTP) protocol. It receives a client request, reads static pages from the filesystem, activates the application server for dynamic pages, and then provides a Hyper Text Markup Language (HTML) file to send back to the client. The adopted standards are:
  - **Uniform Resource Locator** (URL): used for finding web pages.
  - **Hyper Text Murkup Language** (HTML): used for writing web pages.
  - **Graphics Interchange Format** (GIF): for images.
  - **Hyper Text Transfer Protocol** (HTTP): used for client-server interaction.
  - **Transmission Control Protocol Over Internet Protocol** (TCP/IP): communication protocol for data transfer.

- **Application server** is the layer between the client browser and the data residing on a database. It generates dynamic pages, manages the site business logic, and implements the session mechanism. The adopted standard are:
  - **HTTP**: POST or GET with query for sending user-specified data.
  - Integration of a programming language accessible to web server.
  - **Cookies** for storing the state of the session.
- **Database server** stores data on which the application server works and executes queries issued by the application server (updates data, inserts data, and others). There are two types of databases:
  - **Structured Query Language** (SQL)
  - **Not Only SQL** (NoSQL)

The **client-side** is characterized by the browser. The requirements for the client-side are:

- A programming language accepted by all browsers.
- A program embedded in the web page.
- an execution engine in the browser.

The standard used to represent a web page is:

- **Document Object Model** (DOM): to access to on-the fly modification of a web page.
- **JavaScript** (JS): to handle runtime environment on the browser.
- **Cascading Style Sheets** (CSS): modify the aspect on web page.

A web application could be **static** or **dynamic**. A static web application has fixed content that does not change, regardless of who accesses it or when it is accessed. The content is stored in plain HTML files on a server and is delivered to the user's web browser as is. A dynamic web application, on the other hand, generates its content on the fly based on user input, the current date and time, or other factors. The server-side script generates the content, such as Personal Home Page (PHP), Ruby, or Python, delivers it to the user's web browser as HTML. In particular, simple websites often use static web applications that display information and don't require frequent updates, such as personal blogs, portfolios, and informational websites. They are fast, scalable, and easy to maintain, as the contents are stored as plain files on a server and served to the client without any additional processing.

Some **benefits** of using a **static web application** include the following:

- Fast loading times.
- Good performance even under high traffic.
- Cost-effective hosting.
- Increased security as there is no need for server-side code execution.

However, static web applications have limited functionality and cannot support dynamic features such as user authentication, database integration, or server-side processing. In such cases, a dynamic web application would be more appropriate.

2.1.3.2 JavaScript and WebAssembly

Inference, in DL, refers to using a trained model to make predictions on new, unseen data. It involves feeding the model input data and using the learned parameters to produce an output prediction. The DL inference on the browser is one of the exploitations of edge computing. Some benefits of making DL inference on the browser are:

- **Privacy**: by keeping the data and computation on the user's device, DL inference in the browser protects user privacy and reduces the risk of data breaches.
- **Cross-platform**: web browsers run on multiple platforms and devices, so a DL model that performs inference in the browser can be used on any device with a web browser.
- **Reducing server load**: by running the inference on the client-side, the load on the server is reduced, which can improve the system's scalability.
- **Customization**: by running the inference on the client-side, the model can be customized to the user's device and browser, which can improve the performance of the model.
- **Ease of deployment**: making inference in the browser eliminates the need for complex server infrastructure, reducing the cost and effort required to deploy a DL model.
- **Real-time performance**: the performances of DL inference in the browser can be optimized for real-time use cases, such as image classification or object detection, where fast response times are crucial.
- **Offline capability**: browsers can cache DL models, allowing for offline inference even when a user is not connected to the internet.

In order to perform DL applications on the web browser, are implemented several **JS** libraries. The starting point of the process is the representation of DL models. Web applications do not support Pytorch or other model representations, is necessary to think of a more straightforward model representation. Models, data, and inference processes are managed by JS libraries. According to the task assigned and the features supported by the device, different technologies of the **backend** are used: CPU or GPU. For using CPU as backend technology, a component that allows a web application to run ML models with a speed near to the native application is **WebAssembly** [14]; a low-level assembly-like language with a compact binary format that running with near-native performance and provides languages such as CC++, C#, and Rust with a compilation target, and it can run alongside **JS** for working together. Using JS WebAssembly APIs, it is possible to load WebAssembly modules and share functionality with JS. It allows using server-side code on the client-side in the browser. The goals of WebAssembly are:

- **Fast, efficiency and portability**: it can be executed at near-native speed across different operating systems and instruction set by taking advantage to the common hardware capabilities [15].
- **Readability and debugability**: WebAssembly is a low-level language, but his format is readable by human. For this reason is easy to debug its code by hand.
- **Security**: WebAssembly is implemented to be executed in a safe environment. Due to its usage, it is forced to use the same-origin policies.
- **Coesistency with other component**: WebAssembly is implemented with working with other web components; for this reason, it does not break the web.

The web platform can be regarded as consisting of two parts:

- A Virtual Machine (VM) that runs the web application code, such as JS code that power the application.
- A set of Web APIs that the Web application can call to control Web browser/device features and make them work (DOM, CSSOM, WebGL [16], IndexedDB, Web Audio API, etc.).

In the past the VM is used for loading only the JS code. It solved many problems on the web, but for applications that require more computational power, only JS is not enough. Another problem arises with the downloading of very large JS applications; in some cases, it could be prohibitive. Mobile and other devices with

limited resources could increase this bottleneck phenomenon. WebAssembly is thought of as a language that acts together with JS. In this way, the weaknesses of each are filled by the other.
The different types of code can call each other as needed: the WebAssembly JavaScript API allows exported WebAssembly code to be wrapped with JavaScript functions, making them callable in the same way as normal JavaScript functions. Similarly, WebAssembly code can import and call normal JavaScript functions synchronously. WebAssembly code is organized into modules, which are similar in many ways to ECMAScript (ES) modules, making it easy to work with both types of code in a complementary manner.
The key concepts of WebAssembly are:

- **Module**: it represents a binary WebAssembly that is compiled by the browser in executable machine code. A module could be explicitly shared between windows and workers (through `postMessage()` message). A module declares imports and exports just like an ES module.

- **Memory**: a resizable ArrayBuffer that contains the linear array of bytes tath are read and written of WebAssembly instructions.

- **Table**: a resizable typed array that contains the references (e.g to functions), that can not be stored in the memory device for portability and security issues.

- **Instance**: a module is paired with all the states that are used in the execution phase, e.g the memory, the table a set of imported values.

There are many ways to implement WebAssembly in a web application. The simplest uses AssemblyScript [17]. It compiles strict variants of TypeScript to WebAssembly, allows users to use tools such as ESLint, Prettier, and others.
Previously, are cited some APIs that WebAssembly uses. It uses the GPU acceleration capabilities of modern web browsers to perform computationally intensive operations, such as matrix operations and activation functions, faster. The APIs implemented for this scope are:

- **WebGL** [16]: is a JavaScript API for rendering interactive 3D graphics in web browsers. It is based on OpenGL ES, a low-level graphics library for mobile and embedded devices, and provides a way for web developers to create 3D graphics, animations, and games without installing any plugins or software. It provides a set of JS functions for creating 3D graphics and animations, which are then translated into graphics commands and sent to the GPU for rendering. It is widely supported by all major browsers, and it does not require any plugins or software installations.

- **WebGPU** [18]: is a new graphics and computing API for the web that is designed to replace WebGL. It provides a low-level, efficient, and flexible way to perform graphics and computation on the GPU from within a web browser. WebGPU is based on the latest graphics hardware capabilities and is designed to take advantage of modern GPUs, providing improved performance and features compared to WebGL. It is expected to become widely supported in the near future, providing a modern and efficient way to perform graphics and computation in the browser.

The notion of background necessary to know to understand the functioning behind an AI web application was explained. The following section will describe some applications that use the technologies mentioned above.

2.2 Related works

During these years, several applications that are relevant in the edge computing context and, in particular, for **DL web applications**, are developed. Most of them are published and widely used. For example, there is MLitB [19], a ML framework written entirely in JavaScript and able to perform large-scale distributed computing with heterogeneous devices, TensorFlow playground [20], an interactive platform that teaches the fundamentals of DL, and others. Several JS libraries are developed, to implement web applications that exploit DL. Synaptic [21] was created in 2013 by Nathan Rabault. It is one of the earliest DL frameworks for JS. It was designed to provide simple and intuitive API for training and developing neural networks in JS. It was a low-level library that provided a flexible and modular architecture for building and training neural networks, as well as a set of pre-trained models for common tasks. Despite its early promise, it has not been actively maintained in recent years. ConvNetJS [22] was created in 2013 by Andrej Karpathy. As synaptic [21], it is one of the earliest JS libraries for DL implemented. The entire library is based on the transformation of 3-dimensional volumes of numbers. It supports common models of CNN networks and cost function for classification and regression. It has not been active in recent years. Keras.js [23] was created in 2016 by Google's PAIR (People + AI Research) division. It was based on the popular Python library Keras and provided an API for loading and running pre-trained models in the browser. WebDNN [24] was first released in 2017 by the University of Tokyo and Preferred Networks. It claims to be the fastest DL framework in the browser. It supports only inference and all the backends. TensorFlow.js [25] was released in 2018 by Google. It is the successor of deeplearning.js, which is now called Tensorflow.js Core. It is based on WebGL [16] and supports all Keras layers. The most famous JS libraries are cited above. A lot of the following web application use or use them. Different fields are relevant for DL edge computing

applications that exploit the browser. In computer vision, the relevant tasks are image classification and object detection. Significant results there are in video surveillance, object counting, image recognition and others. Teachable Machine [26] is a web application that allows users to teach, to machine, how to respond when they pose a gesture using the camera in the browser. Morphcast [27] is a web application that combines interactive videos and face recognition with emotion, gender and age to create adaptive-media. BeeMachine [28] is a website that identifies bumble bee species. Users received the top three predictions with their probabilities after sending an image of a bumble bee. This website can be used on both mobile and desktop browsers. Due to the pandemic emergency, of such applications are developed, an example is WearMask [29]. It is a web application that understands if a user doesn't wear, wear or wear a mask improperly. It is useful in places like hospitals, schools and any places in which masks are required. Another example of application that is widely used in pandemic times (not only) is WebRTC-based Video Conferencing System [30], that could be used from any device such as laptops, smartphones, tablets and similar. It applies the DL in web browser environment to overcome the limitations of videoconferencing systems background. In Natural Language Processing (NLP), relevant applications are speech translation, speech recognition, sentiment analysis [31] and soon. An example of edge devices that exploit these algorithms, is a vocal assistant. Augmented Reality and Virtual Reality bring AI community attention to web browser latency, energy consumption, bandwidth occupancy issues and in general bad user experience. For these reasons new approaches to collaborative computing [32] improved the results in this field, especially in 5G era. In the audio recognition field, an example is Google search engine for songs; another is Essentia [33] that is a collection of pre-trained models for music-related tasks on the web. Internet of Things (IoT) is the field that brought the most AI attention during the last years. Recently, a new paradigm, called Artificial Intelligence of Things (AIoT) [34], is emerged for the necessity to link AI with IoT, and for the strong demand to integrate AI and edge computing born the Edge Intelligence [13]. Examples include human activity recognition from wearable sensors, pedestrian traffic in a smart city, and electrical load prediction in a smart grid.
The application cited above are a few of the many developed in recent years. There are other fields of edge computing application, but the applications cited above are inherent to this work thesis.

Chapter 3

Methodology

This chapter will present the methodology, i.e., the flow followed to deploy a **static site** that exploits DL in the browser. Figure 3.1 shows the general flow followed to implement the static site able to classify images on the browser. The phases described in this chapter are: **data collection and preparation**, **training**, **ONNX conversion** and the implementation of **static site with ONNX Runtime** for the deployment of ML models.

3.1 Data Collection and Preparation

For ML and DL applications, is fundamental to collect and prepare data suitable for the assigned task. Data are collected for the purpose to **fit** models. For the assigned task, datasets must contain **labels** and low-resolution images to perform a **classification** task on images snapped from different devices, with the light condition and camera performance sub-optimal. There are many ways to collect data [35], the simplest one is to search a dataset on the web, which must be as close as possible to the task of the application to deploy. In this case, the search engines used to search datasets are **Google Scholar** and **Kaggle**.
After choosing the proper dataset, it is loaded on a remote machine through **Secure Socket Shell** (SSH) protocol with a sufficiently large GPU memory to train models.
After finding the proper dataset, data are prepared for the training step. The first thing to do is create **train**, **validation** and **test** set. Usually, the train set is bigger than the others. The proportion used is typically 80:20 (or 70:30 if the dataset is relatively small) for respectively train and test sets. The same proportion is used for train and validation. In the same cases, if data are few, the validation set is not used. Then, data are transformed into tensors by making resizing and center crop, choosing channels, and normalizing with the right mean and standard deviation.

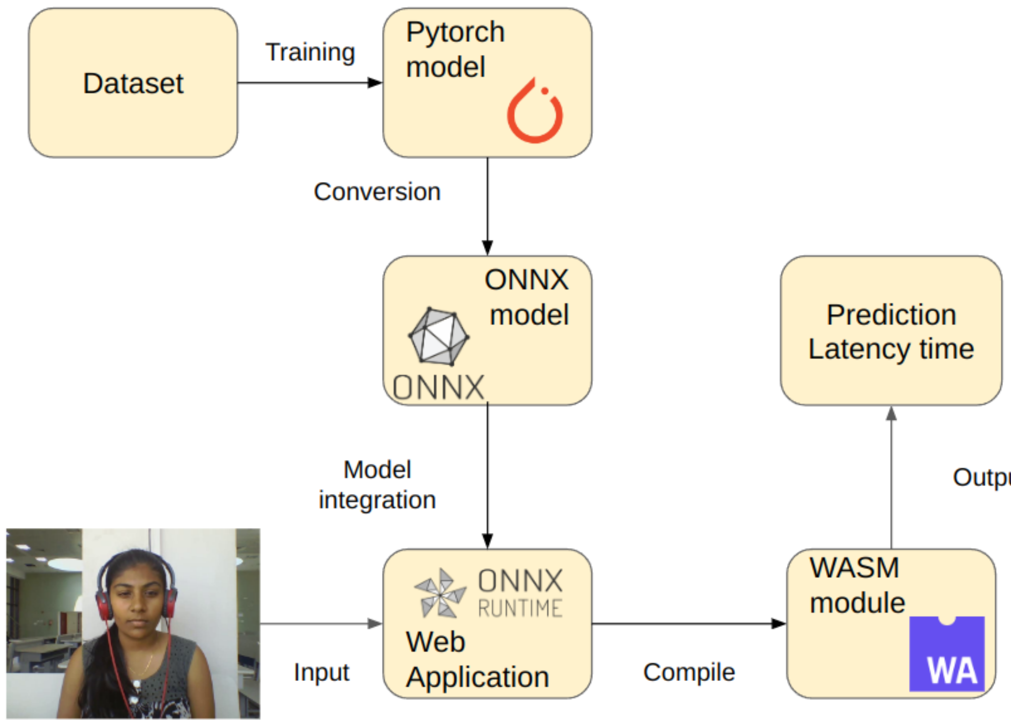


Figure 3.1: Methodological flow followed to implement a Web Application able to do image classification tasks on browser.

For the training set, usually **augmentation** techniques are used, for preventing overfitting.
These transformations are done when datasets are transformed in **dataloader**. This is the final step. Each set is divided into batches, that will be loaded into the memory of the device (GPU in this case) during train, validation, and test phases.

3.2 Training

The development environment used is *Visual Studio Code* [36]. Through it, it was possible to create a folder in which it was created a virtual environment with *Python v3.9*. The DL framework used for this Computer Vision application is **Pytorch**.
After preparing data, the following step is to prepare a set of **hyperparameters** and the other useful **component** for this part:

- **Learning Rate**: is a parameter used in optimization algorithm that determines how much moving towards a minimum of a loss function in each iteration.

- **Epochs**: in one epoch, data makes a complete run. In one of this, data pass exactly one time.
- **Batch Size**: is the samples numbers processed before the update of a network.
- **Step Size**: how many epochs pass before decreasing the learning rate if using a step-down policy.
- **Gamma**: multiplicative factor for learning rate step-down.
- **Model**: the algorithm used for predictions. Here, it is fundamental to choose model and modify it with correct number of classes that our classification required. Weights could be initialize randomly, but in order to optimize resources and time training, it is used a finetuning of the network, i.e. using weights of a previous deep learning algorithm with another similar problem.
- **Loss function**: exist several loss functions, such as Cross Entropy, Mean Squared Error and so on. According to the problem, it is possible to choose one of them, for convolutional neural networks is widely used Cross Entropy Loss 2.4.
- **Optimizer**: is an algorithm or a function that modify weights and learning rate in order to minimize the loss function. It is possible choose which parameters of a network to optimize, with transfer learning only fully connected ones.
- **Learning Rate Scheduler**: is a predefined framework that modify the learning rate during epochs.

At the end of each epoch, the loss is computed on the train set and accuracy (or other metrics) on the validation set to save the model, with certain hyperparameters, that achieve the best performances.
Finally, the model is checked on the test set (to verify if there is overfitting or underfitting) and saved in '`.pt`' format, for reusing it.

3.3 ONNX Model

The format used to run models on a web application is **Open Neural Network Exchange** (ONNX) [4], an open format built to represent ML models. It can be compared, to a programming language specialized in mathematical functions. It defines a set of **operators**, the building block of a model, and a file format that allows using a model on various frameworks, tools, runtimes, and compilers.
Figure 3.2 shows an ONNX representation of linear regression. It defines a **direct**

graph, in which *X*, *A* and *B* are the inputs, *Y* is the output and *MatMul* and *Add* are the called operators, that operate on inputs, i.e., the results of their parents.

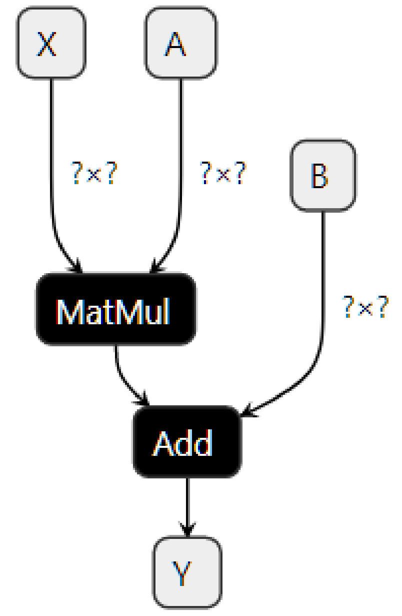


Figure 3.2: Example of a graph representation in ONNX format using Netron [37]. Source [38]

Models can be obtained with different methods:

- From *ONNX Model Zoo* [39]. It is a collection of pre-trained models converted from *Pytorch*, not customized, useful for different tasks.
- From ML frameworks, converting and customizing them.

In this case, it is chosen the second method. Models were implemented and customized using **Pytorch** among frameworks available and then converted with a specific function that allows exporting models defining the operation set, input samples (that coincides with the resize values of models), dynamic axes, input and output names. The package used for converting models from *Pytorch* is *torch.onnx*. Finally, with an appropriate function, it is possible to verify the fairness of the ONNX model obtained.
The features that attract AI community to ONNX are:

- **Interoperability**: the ability to provide a uniform format that acts as an intermediate between ML frameworks.

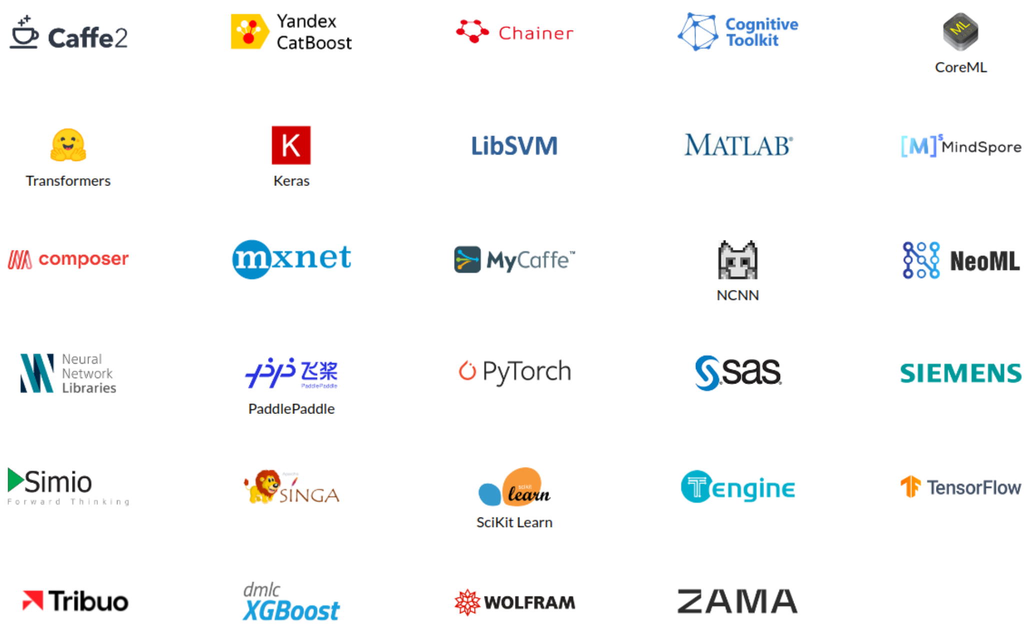


Figure 3.3: Frameworks supported by ONNX for building models.

- **Support for a wide range of models**: support both ML and DL models. It supports a wide variety of neural network architectures and layer types, including feed-forward networks, recurrent networks, and transformers.

- **Portable**: ONNX models can be easily exported and imported, and can be run on a variety of platforms, including Windows, Linux, and Mac, as well as embedded devices and edge devices.

- **Language-agnostic**: ONNX is not tied to a specific programming language, meaning the same model can be used with different languages and tools.

- **Active community**: ONNX is an open-source project with an active community of contributors and users, which helps to ensure its continued development and support.

- **Runtime support**: ONNX Runtime is a high-performance inference engine for ONNX models that can be used across various platforms and devices. It is not the only runtime used to deploy ONNX models.

Figure 3.4: Runtimes used for deploy ONNX models.

3.4 Image Classification inference on Static Site using ONNX Runtime Web

As described in Chapter 2, a web application has two sides: a client and a server. In this case, the use of a **static site** is fundamental for its benefits:

- **Speed**: the content is fixed. There is no need for the server, to generate it on the fly, which means that pages load faster and the website can handle more traffic.
- **Security**: there is no server-side scripting or database connections, there are fewer opportunities for hackers to exploit vulnerabilities.

For the server side **Node.js** v16.15.0 [40] is used, for the generation of **static site** are used **Next.js** v11.1.2 [41] (a **React.js** [42] framework), written in **typescript**. For bundle JS files **webpack** v1.1.4 [43] is used. To make the appearance of the page, are used **HTML** and **CSS**. For performing inference on the client side, for enabling ONNX models to run in the web browser, and for deploying models in a production environment is used **ONNX Runtime**. It is an open-source project developed by Microsoft, a high-performance inference engine for ONNX models designed to be fast, efficient, and flexible. ONNX Runtime is implemented in C++ and runs on various platforms, including Windows, Linux, and MacOS. It supports multiple programming languages, including C++, Python, and C#, and can be used in several environments, including desktop applications, cloud services, and IoT devices. ONNX Runtime is optimized for running ONNX models and provides a consistent interface for inferencing, regardless of the underlying hardware. This makes it easy for developers to conceive ONNX models in different environments

and provides a level of consistency and compatibility across multiple platforms. The goal of ONNX Runtime is to provide a fast, efficient, and flexible engine for running ONNX models, making it easier for developers to build and deploy DL applications in production.
In particular, **ONNX Runtime Web** is used. It is a **WebAssembly** (Wasm) based version of ONNX Runtime. It runs in the browser and provides a high performance inferencing with ONNX models. ONNX Runtime Web is designed to run efficiently on web browsers and enables the deployment of ONNX models on the web, making it possible to perform inferencing directly in the browser without requiring any additional servers or infrastructure. Some key features of ONNX Runtime Web include:

- **Support for ONNX models**: ONNX Runtime Web can run any model that is in the ONNX format, which is a standard format for exchanging neural network models between different frameworks and tools.

- **Cross-platform compatibility**: ONNX Runtime Web can run on a wide variety of platforms, including web browsers on desktops and mobile devices.

- **High performance**: ONNX Runtime Web is optimized for performance, making it suitable for running large and complex models in real-time.

- **WebAssembly support**: ONNX Runtime Web can use WebAssembly [2] for performance improvements.

- **Easy integration**: ONNX Runtime Web can be easily integrated into web applications, allowing developers to add ML capabilities to their apps with minimal effort.

Figure 3.5 describes the component and the steps followed during the whole process. The **input data** in this case is the image snapped from the device in use.
Data is pre-processed and transformed in tensor, then it is passed to **ONNX model**. The flow followed by ONNX Runtime to manage the ONNX model is:

- ONNX Runtime convert the ONNX model graph into an in-memory graph representation.

- It executes a set of graph optimizations (graph-level transformation, such as graph simplification, elimination or complex fusion of nodes, and layout optimizations) independent from the provider used.

- It is divided into a set of sub-graphs based on available execution providers.

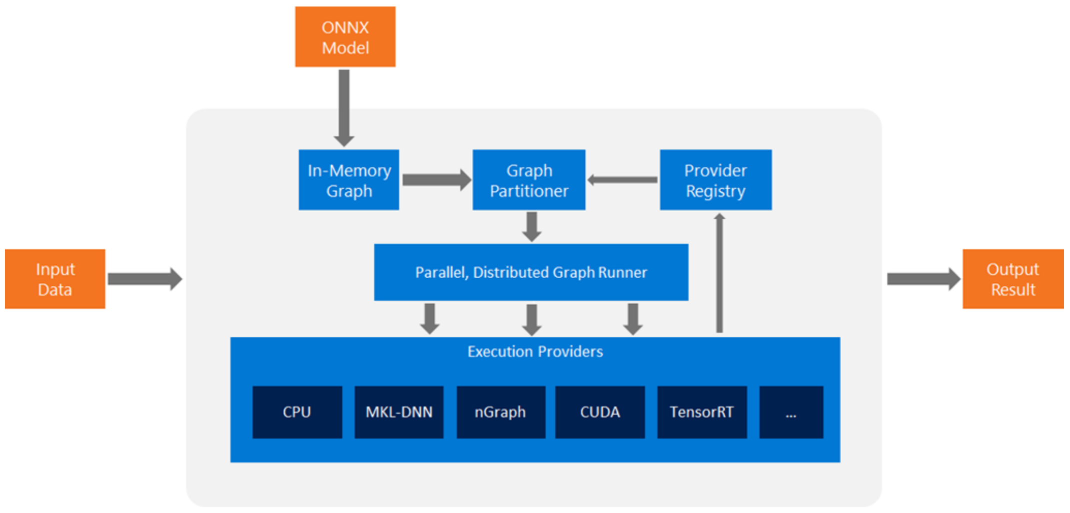


Figure 3.5: High-level system architecture of ONNX Runtime Web. Source [44]

- Each sub-graph is assigned to an execution provider. To be sure that a execution provider could execute sub-graph, is needed to query the capability of the provider using the `GetCapability()` function.

The use of WebAssembly JS API by ONNX Runtime Web is to load a `wasm` module (ONNX model) in a web page. It is useful because there is not necessity of a separate server. The flow of operation made by `wasm` module are:

- WebAssembly module is compiled from source code.
- The compiled WebAssembly module is loaded into the browser as a binary file.
- The browser's WebAssembly engine parses and decodes the binary file into an internal format, ready for execution.
- The WebAssembly module is executed by the browser's WebAssembly engine, running in a separate environment from JS.
- WebAssembly code can interact with the JS code and DOM in the browser by calling JS functions and accessing the DOM, and vice versa.

Finally, the **output result** of the web application implemented is the **prediction** with its **probability** and the **latency time**, time used for prediction.

Chapter 4

Experimental Results

The previous chapter described the methodological flow that it must be follow to deploy a web application for image classification. This chapter discusses results achieved on a real application. The task of this application is to consider the **affective states** of a worker with the purpose of make possible intern statistics of a work environment. The goal of these experiments is to demonstrate how making DL inference on browser is useful in terms of **security** and **portability**. This chapter illustrates the experimental setup adopted and the results obtained after having tested the application with different DL models and hardware settings.

4.1 Experimental Setup

This section introduces the datasets, models, and hyperparameters used in the training phase, the components useful for obtaining ONNX models for the inference phase.

4.1.1 Datasets

In order to test a web application that performs inference on a image classification tasks, two different datasets are considered: Office31 [45] and DAiSEE [46].
These two datasets were chosen because their data are similar to working or smartworking settings. The approach adopted, was to start with a simpler task using Office31 as a test, and then use DAiSEE for the final application.

4.1.1.1 Office31

Office31 is a dataset used for Domain Adaptation. It is a sub-discipline of ML deals of scenario in which a model is trained on a source distribution, and used on a target domain, different but related.

From the last update, this dataset contains **4203** element distributed among **31 object categories**, commonly encountered in a working environment, in three domains: **Webcam**, **Amazon** and **Digital Single-Lens Reflex** camera (DSLR). These 31 objects are commonly used in office settings, such as mouses, laptops, desktops and others.

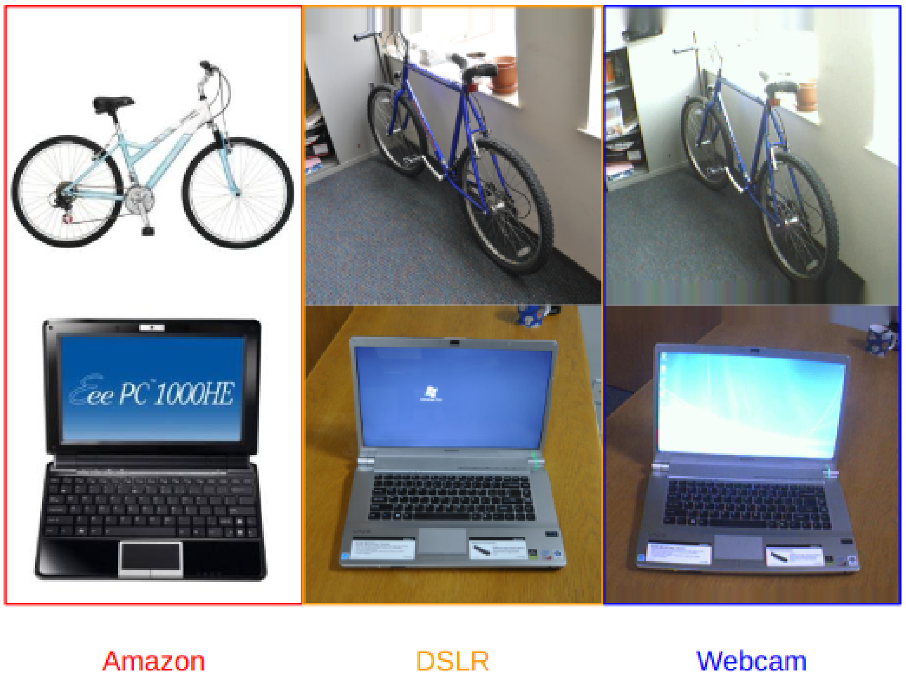


Figure 4.1: Example of a bike and laptop PC in the three domains of Office31 dataset.

Images are distributed among classes in this way:

- **Amazon**: contains on average 90 images per class and in total 2848, captured on online merchants websites. Images are captured on a clean background and with a unified scale.
- **DSLR**: contains 5 images per class and in total 529 low-noise high resolution (4288×2848) ones.
- **Webcam**: contains on average 25 images per class and 826 of low resolution (640×480) images in total, whit significant noise.

As discussed above, this application was designed to be used in a working environment, using mainly images with low resolution. For this purpose, only **Webcam** branch is considered.

4.1.1.2 DAiSEE

DAiSEE is a multi-label video classification dataset that is used to recognize the users' affective states of **Boredom**, **Engagement**, **Frustration** and **Confusion**. For each affective state, there are four levels of labels: ***'very low'***, ***'low'***, ***'high'***, ***'very high'***.

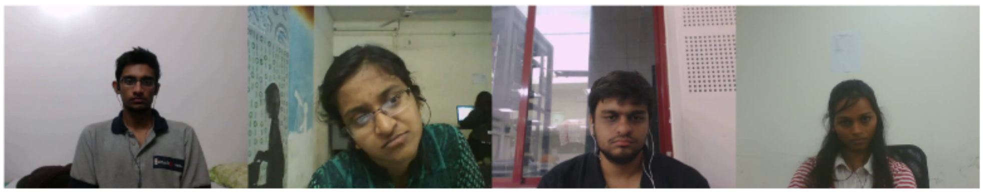


Figure 4.2: Example of frames taken from class "Confusion" of DAiSEE dataset. Starting from the left, there are respectively the four levels of labels.

The dataset captures **"in the wild"**, i.e., in the conditions that are as close as possible to the real world: different environments with different background noise, illumination conditions, head poses, and occlusions.
It has **9068** videos snipped and captured from **112 users**. In table 4.1, we can see how they are distributed samples among classes.

Affective States	**Very Low**	**Low**	**High**	**Very High**
Boredom	3869	2931	1934	334
Confusion	6024	2191	752	101
Engagement	61	459	4477	4071
Frustration	6986	1649	346	87

Table 4.1: Distribution of DAiSEE dataset samples among classes.

In this case, only class **"Confusion"** are considered with two labels: ***"very low"*** and ***"very high"***.

4.1.2 Models

As discussed in Chapter 2 models used are **CNN**.
To solve the problem of **portability** some CNN are chosen with different sizes, allowing, also to devices with low performance, to use the application. The models used are:

- **MobileNetV3 small** and **large** [47]: it is a CNN developed by Google. This model is perfect for mobile or low performance devices. This model is the combination of a **Neural Architecture Search** (NAS) complemented in NetAdapt [48] algorithm, and then improved thanks to an innovative architecture. NAS [49] is a process that tries to make a model that produces output modules, that, together, could make a model that reaches the best accuracy searching among all the possible combinations. This process is the starting point of **NetAdapt** [48] algorithm. In the following steps, NetAdapt generates some *proposals* that must have latency time lower than the previous ones. Then, it sets the weights of the new *proposal* with the previous one and sets random initialisation of any new filters. Finally, it fine-tunes of the selected *proposal* until the target latency is reached. The general architecture is based on some blocks introduced by each network version. MobileNetV1 [50] introduced the *depthwise separable convolutions* replacing the traditional convolutional layers. This block aims to divide spatial filtering (made from *light weight depthwise layers*) from the feature generation mechanism (made from heavier 1×1 *pointwise convolutions*). The MobileNetV2 [51] added the *linear bottleneck* and the *inverted residual structure*. This structure maintain a compact representation in the input and output while it expands internally to a space with higher-features for increasing the expressiveness of nonlinear transformation per channel. MnasNet [52] is built on the structure of MobilenetV2 and it introduces **lightweight attention modules** based on *squeeze* and *excitation* modules into the bottleneck structure. Finally the **MobileNetV3** is the combination of these blocks and layers with the introduction of *swish* non-linearities and the *hard sigmoid* in order to improve accuracy. The difference between **large** and *small* is in the size of model and they are use for high and low-performance resource respectively.

- **ResNet18** and **ResNet34** [53]: it is a CNN that induced a breakthrough in computer vision field. In order to reduce the error rate, networks became deeper, but a phenomenon called **Vanishing/Exploding gradient** occurred. The gradient became zero or tends to infinite. Consequently, the training test and error rate increase. ResNet introduces the **Deep Residual Learning**, the gradients flow through skip connections backwards from later layers to initial filters. These connections are called **shortcuts connection** and which turn a plain network in residual. The *identity shortcuts* (eq. 4.1. y is the input and x is the output vectors of the layer considered. $F(x, \{W_i\})$ is the residual function to learn) are used when input and output have the same dimension. If the dimension became too large are considered two options: shortcuts still performing *identity mapping* with extra zero-padded to increase dimension, or *projection shortcut* (Eq. 4.2 y is the input and x is the output vectors of the layer considered. W_s is the square matrix of linear projection.) matching dimension with convolution 1×1.

$$y = F(x, \{W_i\}) + x \tag{4.1}$$

$$y = F(x, \{W_i\}) + W_s x \tag{4.2}$$

The difference between ResNet18 and 34 is the **depth**; the last one has more layers.

- **Inception V3** [54]: it is a CNN that belongs to Inception family. This network is composed by **Inception modules**, that has different kernel size, and **auxiliary classifiers**. The motivation that brings researchers to implement these networks are the large variety of location of the subject in a picture, the depth of networks that brings to overfitting and the computational expensiveness introduced by large convolutional operations. For these reasons, the architecture is more **"wider"** than deeper. The innovations brings by Inception V3 are: the use of **Label Smoothing** technique, the **Batch Normalization** in the auxiliary classifiers and the factorized 7×7 convolutions. The Label Smoothing is a **regularization** technique; it introduces noise for the labels to prevent overfitting. Researchers notice that the auxiliary classifiers did not contribute in a significant way, until near the end of training phase, when accuracy was nearing saturation. The Batch Normalization reduces the internal covariate shift that produces an acceleration in training phase. It also introduces a benefit for the gradient flow through the network.

- **EfficientNet V2 s**, **m** and **l** [55]: it is a CNN that belongs to the family of networks optimized for FLOPs (Floating-Point Operation per second) and parameters efficiency. It has faster training and better parameters efficiency than the previous versions. To develop this models, it is used a combination of **NAS** [49] and **scaling**, to jointly optimize training speed. NAS [49] is used to make optimal model design choices and find hyperparameters. Scaling strategies are rules that indicate how to make bigger small networks. Training speed is improved with new regularization methods and guidelines to training efficiently. **Progressive learning** is used to accelerate training by progressively increasing the image size. Various types of convolutional and building blocks are used to implement a network. Widely used and fundamental blocks are **MBConv** and **fused-MBConv**. They are used for efficiency reasons, the MBConv is used with depthwise convolution and fused-MBConv is used without depthwise convolution. **Scaling** EfficientNet V2 s produces EfficientNet m and l. The difference among these networks is the size.

ResNet18 is used both for Office31 and DAiSEE datasets, the other networks are used only for DAiSEE.

4.1.3 Data Preparation, Data Preprocessing and Training

This section deals with techniques used in **Data Preparation and Preprocessing** step, and **hyperparameters setup** of **Training** phase, respectively for Office31 and DAiSEE datasets. The main steps that are highlighted in this section are the **split** of the dataset (train, validation, and test set), the **transformation** of data, the choice of **hyperparameters**, and the **Transfer Learning** techniques used. Both training processes have been carried out with **NVIDIA TITAN Xp** GPU with **12 GB** of memory.

4.1.3.1 Office31

The branch **"Webcam"** of Office31 has **795** objects. As shown in Figure 4.3, the dataset is slightly imbalanced; the lowest class size is the class *"ruler"* and contains 11 elements; the largest one is *"monitor"* that contains 43 elements.

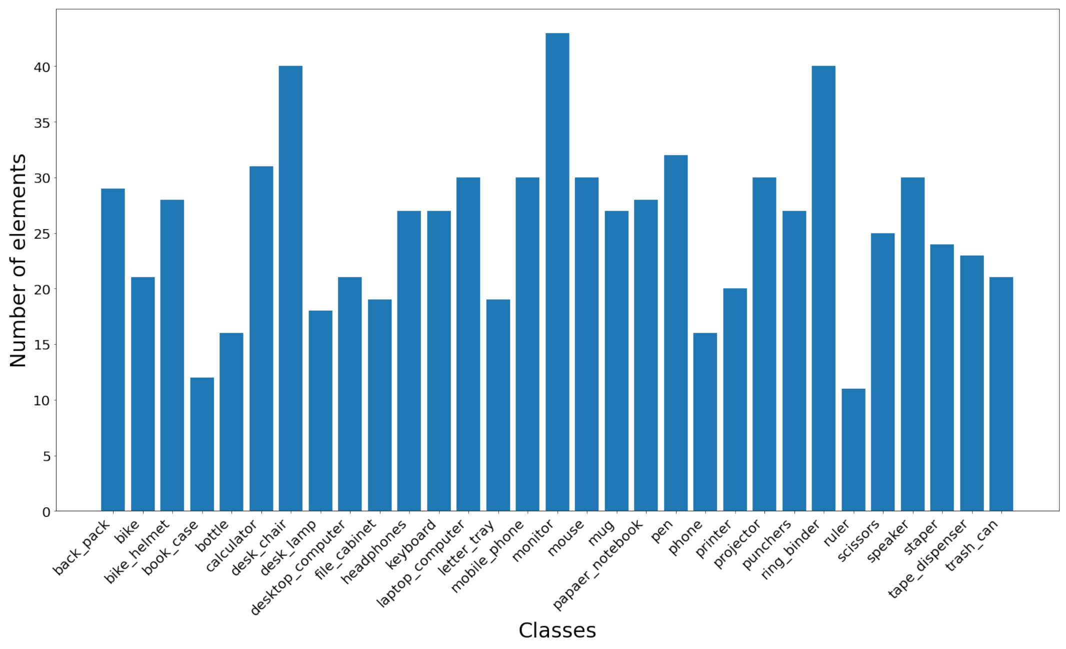


Figure 4.3: Distribution of "Webcam" branch, of Office31 samples among 31 classes.

The resulting subset are:

- **Train**: 332 objects
- **Validation**: 331 objects
- **Test**: 132 objects

After preparing the datasets, **dataloaders** are obtained with a **batch size** of **64** for all sets and with **shuffle** only for the train set. Images are transformed in tensors, resized, and center cropped, respectively **256** and **224** for **ResNet18**. Only for images that belong to the train set was used **data augmentation** technique with *ColorJitter* to prevent overfitting. As mentioned above, the model used is ResNet18. It is pre-trained on *ImageNet* [56] to perform **transfer learning**. The last fully-connected layer is replaced with a linear one with **31** classes, and the images are **normalized** with a mean and standard deviation of *ImageNet*, respectively [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225]. In this case, for better performance and more speed in the training phase, only on the fully-connected layer is performed **finetuning**, e.g., only weights of the last layer are updated.

The **loss** function used is the **Cross Entropy** 2.4. The **optimizer** used is **Stochastic Gradient Descent** (SGD) (algorithm 1) and the input values used are:

- **learning rate**: 0.001
- **weight decay**: 0.00005
- **momentum**: 0.9

Finally, a **scheduler** to decay learning rate is used with a γ value of 0.1 and a **step size** of 10. The number of **epochs** used are 25. In the training process, loss (2.4) and F1 score (2.11) is computed after each epoch. The choice of **F1 score** as evaluation metrics is due to the imbalanced dataset.

4.1.3.2 DAiSEE

The branch of **"Confusion"** of DAiSEE contains **268401** frames. Figure 4.4 shows the distribution of frames among the 4 classes and the imbalance distribution is evident. The largest class is *"very low"* and contains 178860 elements, while the smallest one is *"low"* and contains 3008 samples. In this case, the goal is to detect if a user is **confused** or not, hence, only *"very low"* and *"very high"* class are considered.
Data are taken from *Activeloop* website [57], converted from tensors to images, stored on the device, and finally re-transformed to tensors. Data had already been divided into three sets. After filtering and taking only the two classes mentioned above, the results are shown as follows:

- **Train**: 149596
- **Validation**: 46221

Algorithm 1 Stochastic Gradient Descent algorithm.

```
inputs: $\gamma$ (learning rate), $\theta_0$ (parameters to optimize), $f(\theta)$ (objective), $\lambda$ (weight decay), $\mu$ (momentum), $\tau$ (dampening), nesterov, maximize
  for $t = 1$ to . . . do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\lambda \neq 0$ then
      $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    end if
    if $\mu \neq 0$ then
      if $t > 1$ then
        $b_t \leftarrow \mu b_{t-1} + (1 - \tau)g_t$
      else
        $b_t \leftarrow g_t$
      end if
      if nesterov then
        $g_t \leftarrow g_t + \mu b_t$
      else
        $g_t \leftarrow b_t$
      end if
    end if
    if maximize then
      $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
    else
      $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$
    end if
  end for
  return $\theta_t$
```

- **Test**: 47288

Afterwards, **dataloaders** are obtained with a **batch size** of **32**. Only for the train set, **shuffle** techniques and a strategy to draw samples from the dataset are used. The second technique assigns **different weights** to samples based on the origin class because the two classes are imbalanced. Images are transformed in tensor and preprocessed using values in table 4.2. For the train set **Data Augmentation** is used with *RandomRotation* of 20° and *ColorJitter*. Each model is pre-trained on *ImageNet* to perform **transfer learning**. The last fully-connected layer is replaced with a linear one with **2** classes and images are **normalized** with a mean and standard deviation of *ImageNet*, respectively 0.485, 0.456, 0.406] and [0.229, 0.224, 0.225]. Also in this case, for the same reasons explained for Office31 training, only

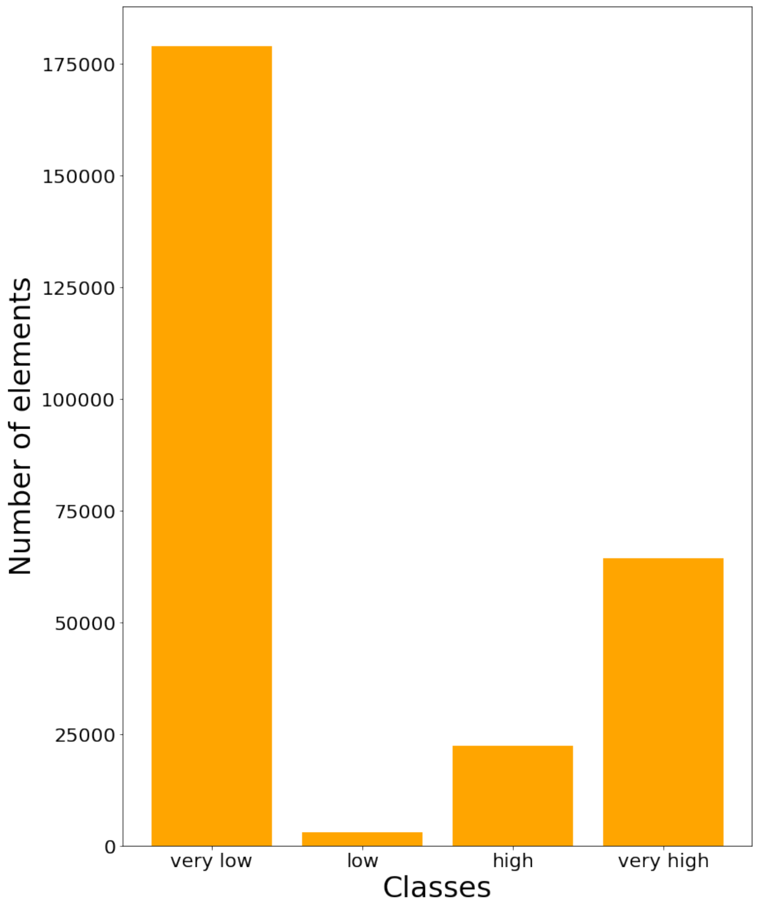


Figure 4.4: Distribution of branch "Confusion" of DAiSEE dataset frames among the 4 classes.

on fully-connected layer is performed **finetuning**.

The **loss** function used is **Cross Entropy** 2.4. The **optimizer** used is **Stochastic Gradient Descent** (SGD) (1) and its input values are:

- **learning rate**: 0.001
- **weight decay**: 0.00005
- **momentum**: 0.9

A **scheduler** to decay learning rate is used with a γ value of **0.1** and a **step size** of **10**. The number of **epochs** used is **50**. During training process, after each epoch, loss (2.4) and **F1 score** (2.11) are computed. The choice of this metric is due to the imbalance of dataset.

Models	Resize	Center Crop
Resnet18-34	256	224
Inception V3	342	299
EfficientNet V2 s	384	384
EfficientNet V2 m-l	480	480
MobileNet V2 large-small	256	224

Table 4.2: Values of resizing and cropping used with each model on DAiSEE samples.

4.1.4 ONNX format conversion and Static Site deployment

Models trained, for both DAiSEE and Office31, are saved in `.pt` format and then converted in `.onnx` format with an **operation set** version of **16**. The conversion is carried out with **NVIDIA TITAN Xp** GPU of **12 GB** of memory. The browser used to deploy the static site is **Google Chrome**.

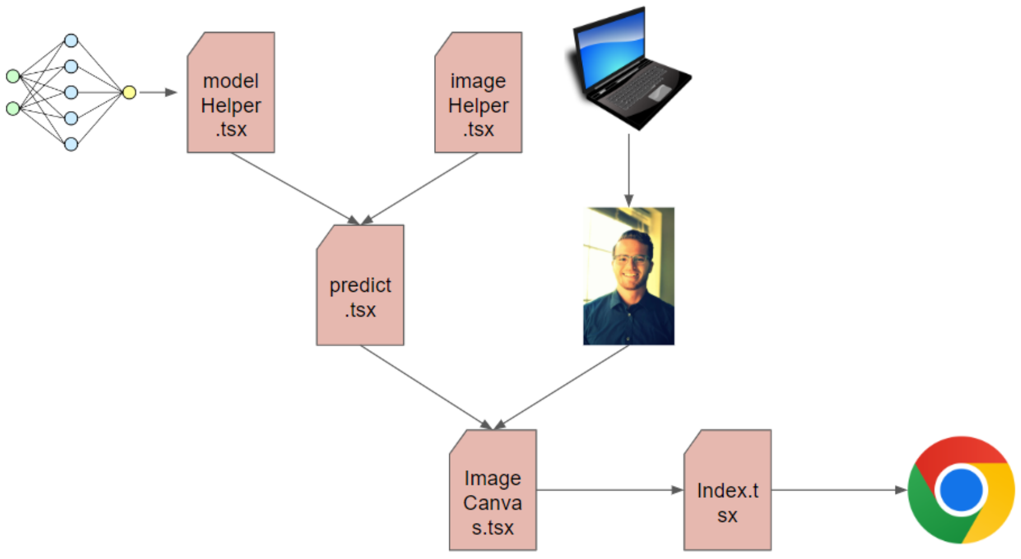


Figure 4.5: The most important components of web application.

The main Typescript files handle the part of **inference**:

- ***imageHelper.ts***: image is loaded from *public* folder of the project and transformed in a tensor. Then, data are preprocessed according to the resize and centre crop values of the model chosen.
- ***modelHelper.ts***: the tensor created in the previous step is ready for inferencing. An inference session through *ort.InferenceSession.create()* is created. The

fundamental parameter passed to this function is `'wasm'` (CPU) as execution provider (for better compatibility of operators) and the model previously converted in *.onnx* format. In each inference session a WebAssembly module is used to run the model as near-native speed. At the end of each inference session, the time to do it, the prediction and its probability are saved.

- ***predict.js***: contains all the functions written in the previous files.

Data URLs used for inference and the identification of classes are saved in *data* folder. The web component that has the buttons and display elements is *Image-Canvas.tsx*. The functions written *predict.js* are called in this component. Finally, this component is called by *index.tsx*. It is in charge of the title and the **Image-Canvas.tsx** components of the web page. Figure 4.6 shows the final aspect of the two web application.
The last two important files are *next.config.js* and *package.json*. The first file is the webpack configuration implemented in the NextJS Framework. Its task is to copy *wasm* files and model folder files to the *out* folder for deployment. The second file contains the assets needed for the static site deployment and puts them in *out* folder.

4.2 Results

This section shows the results achieved with both datasets using the web application developed. The experiments have been done with a combination of different DL **models** and **hardwares**. The two crucial **metrics** used for judging the goodness of a model for the device used to run the application are: **ONNX model size** and **latency time**, the time used for a single prediction. The experiments are done with devices described in Table 4.3. Three type of devices are taken into consideration: a **high-performance PC**, a **low-performance PC** and a **smartphone**. The browser used to run experiments is **Google Chrome** for all devices.

4.2.1 Office31

Table 4.4 shows the results obtained with *Device 1* (features in Table 4.3). Means and standard deviations **latency** are computed by taking 10 images from the Google search engine.
This experiment aims to test if integrating of the ONNX model with the web application works and if the predictions are correct. The latency time is acceptable because the device used has good hardware features. Due to the simplicity of the task, **almost all** predictions are **correct**.

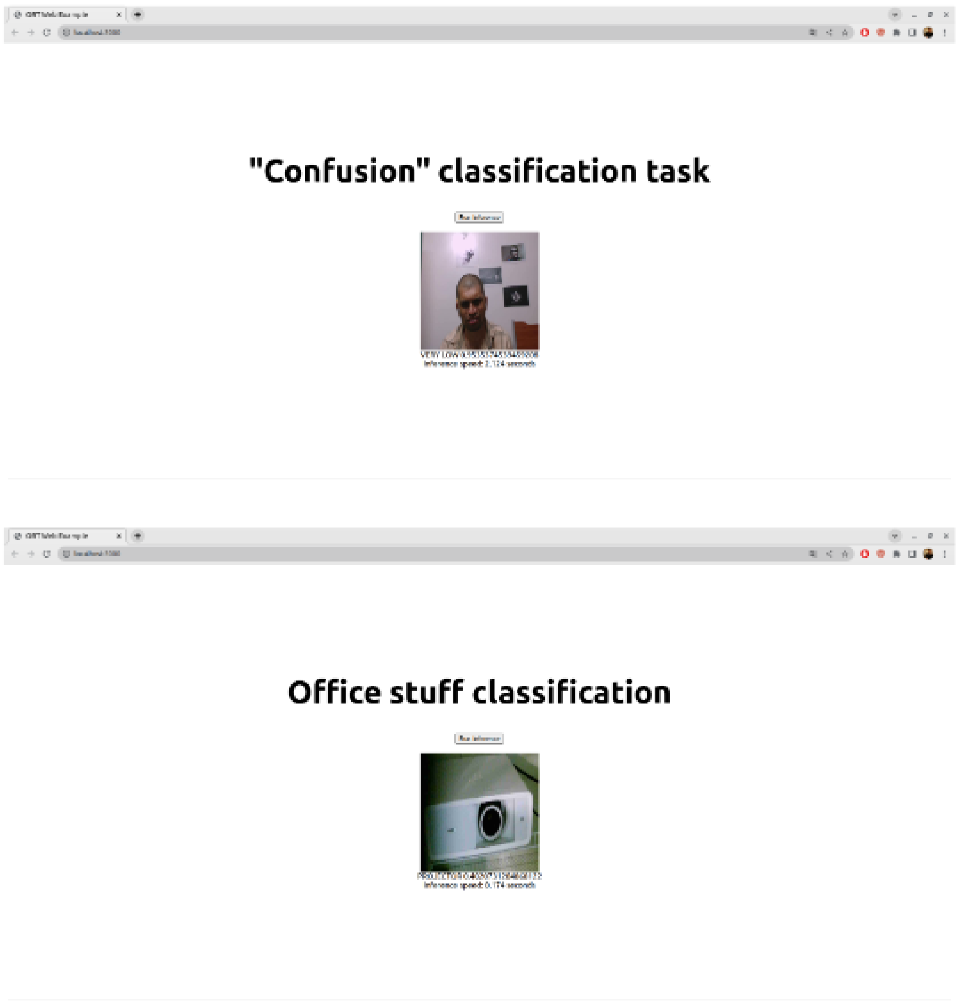

"Confusion" classification task
Run inference
Inference speed: 2.174 seconds
Office stuff classification
Run inference
PROJECTOR
Inference speed: 0.174 seconds

Figure 4.6: Static sites appearance.

4.2.2 DAiSEE

Table 4.5 refers to the results obtained with all different combinations of models and hardware. The three critical indicators considered for the comparison are the **latency** (inference time in browser) computed in ms, the **ONNX size** of the model computed in MB, and the **F1 score** of Pytorch models. The mean and standard deviation of latency are computed by taking ten images (five from class *"very low"* and *"very high"*) from the test set used to test Pytorch [6] models. The same images are used for all devices used in this experiment. The F1 score of Pytorch models is computed in the test phase before the ONNX format conversion.

Device	Model Name	Model CPU	Clock rate	Core CPU	RAM
1	Asus VivoBook X571GT	Intel® Core™ i7-9750H	2.60 GHz	12	16 GB
2	Asus X55A-SX093D	Intel® Pentium® B980	2.40 GHz	2	4 GB
3	Redmi Note 10 5G M2103K19G	Octa-core Max	2.20 GHz	8	4 GB

Table 4.3: Hardware features of devices used to test the web application developed. Device 1 and 2 are notebooks, the Device 3 is a smartphone.

Model	Latency (mean - std)	ONNX size	F1 score Pytorch
Resnet18	(155.56 - 2.70) ms	42.68 MB	93.93%

Table 4.4: Result with Office31 dataset obtained with Device 1.

From the experiments conducted, the last network (**EfficientNet V2 l**) resulted in too heavy for Devices 2 and 3 due to the limited performance of both devices. Figure 4.7 shows the relationship between the mean of latency time and the size of each ONNX model. There are three lines, one for each device. It is possible to identify two general behaviors:

- The latency time grows with the decrease in the device's performance.
- The latency time grows with the increasing size and complexity of the ONNX model.

We can see how the small models are suitable for all devices in this case, the latency time computed with the **MobileNet V2 small** and **MobileNet V2 large** is similar for all devices.
The **F1 score** achieved by Pytorch models is a relevant element to consider. In this case, the F1 scores obtained from each model are very similar. The higher F1 achieved is **92,21%** from **EfficientNet V2 m**, but it has latency time and size too large compared to the others. The two best alternatives are **MobileNet V2 small** and **Inception V3** with respectively **91,32%** and **91,95%** F1 score. These two models have lower latency time and size.

	Latency (mean - std)				
Models	**Device 1**	**Device 2**	**Device 3**	**ONNX size**	**F1 score Pytorch**
MobileNet V3 small	(32,67-6,76) ms	(31,33–1,32) ms	(122,33–26,41) ms	5,76 MB	91,32%
MobileNet V3 large	(74,78-3,70) ms	(88,89–5,49) ms	(187,33–13,96) ms	16,01 MB	89,15%
ResNet18	(314,44-6,04) ms	(422,33–78,04) ms	(768,22–17,93) ms	42,63 MB	89,76%
ResNet34	(629,44-15,74) ms	(765,78–2,39) ms	(1474,44–59,02) ms	81,18 MB	81,95%
Inception V3	(918,33-14,83) ms	(1152,11–1,45) ms	(2374,33–243,55) ms	83,09 MB	91,95%
Efficient-Net V2 s	(1500,78-52,95) ms	(1971,33–112,54) ms	(3493,00–85,27) ms	76,75 MB	88,48%
Efficient-Net V2 m	(4410,67-109,93) ms	(5703,89–119,05) ms	(10322.44–250.45) ms	201,18 MB	92,21%
Efficient-Net V2 l	(9680,56-188,11) ms	-	-	446,37 MB	89,15%

Table 4.5: Result of experiments on DAiSEE dataset using different devices and models.

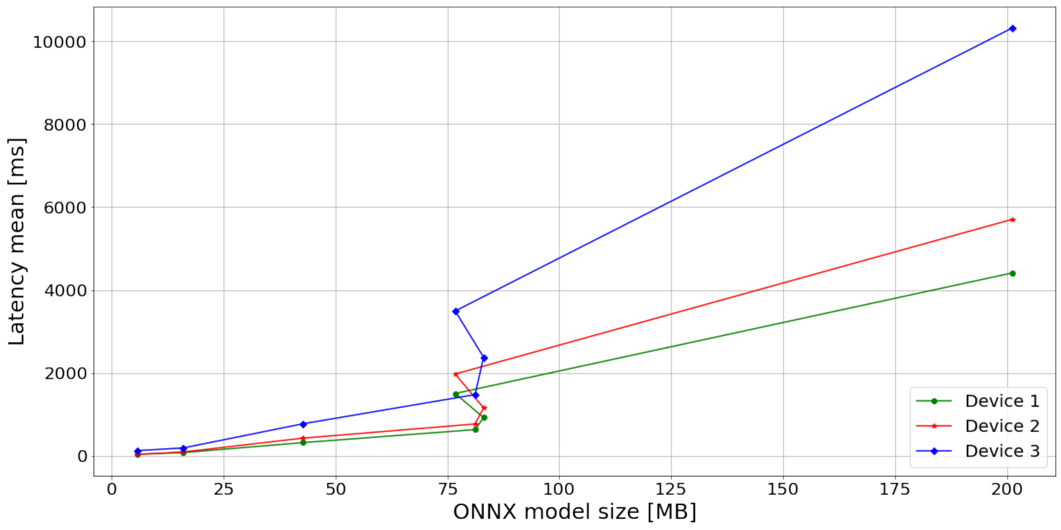


Figure 4.7: Relationship between the latency mean of models and their size.

Chapter 5

Conclusion

In this present master thesis is discussed the methods for exploiting the browser to ensure the **privacy** and the **portability** of a DL application. There are several ways to ensure **privacy** and **portability**, but WebAssembly and JS libraries are a good compromise between the easiness of implementation and the goodness of the performance.

The web application developed, has a simple layout. The predictions obtained with ONNX models on Office31 are all right, while with DAiSEE there are some prediction errors. This difference in the performance could be due to the different complexity between the two datasets. The networks implemented are simple and not provided of other branches. The choice of this implementation design, is due to the scope of this thesis work. The size and the complexity of the networks are important features that allow also to low-performance devices to make DL inference. There are techniques able to simplify the complexity and able to reduce the size of a network. ONNX Runtime provides a tool to use **Quantization** method on ONNX models. The basic idea is to represent the weights and activations of a DL model using fewer bits than their original precision. As shown in Figure 5.1, larger networks have a larger inference time. Thus optimization technique cited above could be useful for them.

Figure 5.1 shows the latency mean of ONNX and Pytorch models used in web applications. This test was made only on Device 1 4.3. Table 5.1, shows the results obtained from the test. Figure 5.1 shows the gap between the latency time of the two model format. The latency gap between the two model format increases with the growth of the complexity of the model. The larger gap lies between **Resnet-34**, with a ONNX model **18.32** times slower than Pytorch one. The smallest gaps lies between **MobilenetV3 small** and **large**, with ONNX model **1,96** and **3,39** times slower than Pytorch model respectively. **MobilenetV3** is optimize to run on low-performance device, in fact the gap observed is very small respect to the others. For the other models, techniques of optimizations are necessary.

Models	**Pytorch Latency (mean - std)**	**ONNX Latency (mean - std)**
Resnet18	(18,85 - 0,22) ms	(314,44 - 6,04) ms
Resnet34	(34,36 - 5,40) ms	(629,44 - 15,74) ms
InceptionV3	(70,88 - 8,03) ms	(918,33 - 14,83) ms
MobileNetV3 small	(16,60 - 10,84) ms	(32,67 - 6,76) ms
MobileNetV3 large	(22,03 - 2,58) ms	(74,78 - 3,70) ms
EfficientNetV2 s	(127,30 - 23,14) ms	(1500,78 - 52,95) ms
EfficientNetV2 m	(302,43 - 13,05) ms	(4410,67 - 109,93) ms
EfficientNetV2 l	(566,73 - 10,31) ms	(9680,56 - 188,11) ms

Table 5.1: Results of latency time obtained on the same pictures using ONNX models and Pytorch models. For this comparison, models are tested on Device 1 (table 4.3).

The web application developed is not able to switch models according to the hardware performance. To implement the process of switching models according to the hardware of the device, some ideas can potentially be helpful for future works:

- `Navigator.deviceMemory` [58]: a read-only property of the JS `Navigator` interface. The value returned is an approximation of the **amount of memory** (in gigabytes) available to the device and can be used to make informed decisions about the performance of the device and the resources that could be allocated to a web application. This value can be used to optimize the performance of a web application by making adjustments to the way the application uses resources. For example, if the value indicates that the device has a low amount of memory, the application can reduce the resources it uses, such as images and animations, to ensure that it runs smoothly. In this context, the resources are the networks. Unfortunately, this property has an **experimentation status**; it can not be used in the building phase.
- **Choose your model**: implementing a **menu** that allows users to choose the right network according to the performance of his/her device.

This Master thesis aims to study and implement technologies that allow doing, DL inference on all devices and operating systems. It confirms that using web application for DL is **safe** and **portable**. DL models have to be light and straightforward to allow the correct function and also to more powerful hardware. WebAssembly is a relevant technology for speed up performances of DL models in a web application. In particular, ONNX Runtime Web is a good solution for implementing DL in the browser, but using more extensive networks without any optimization process,

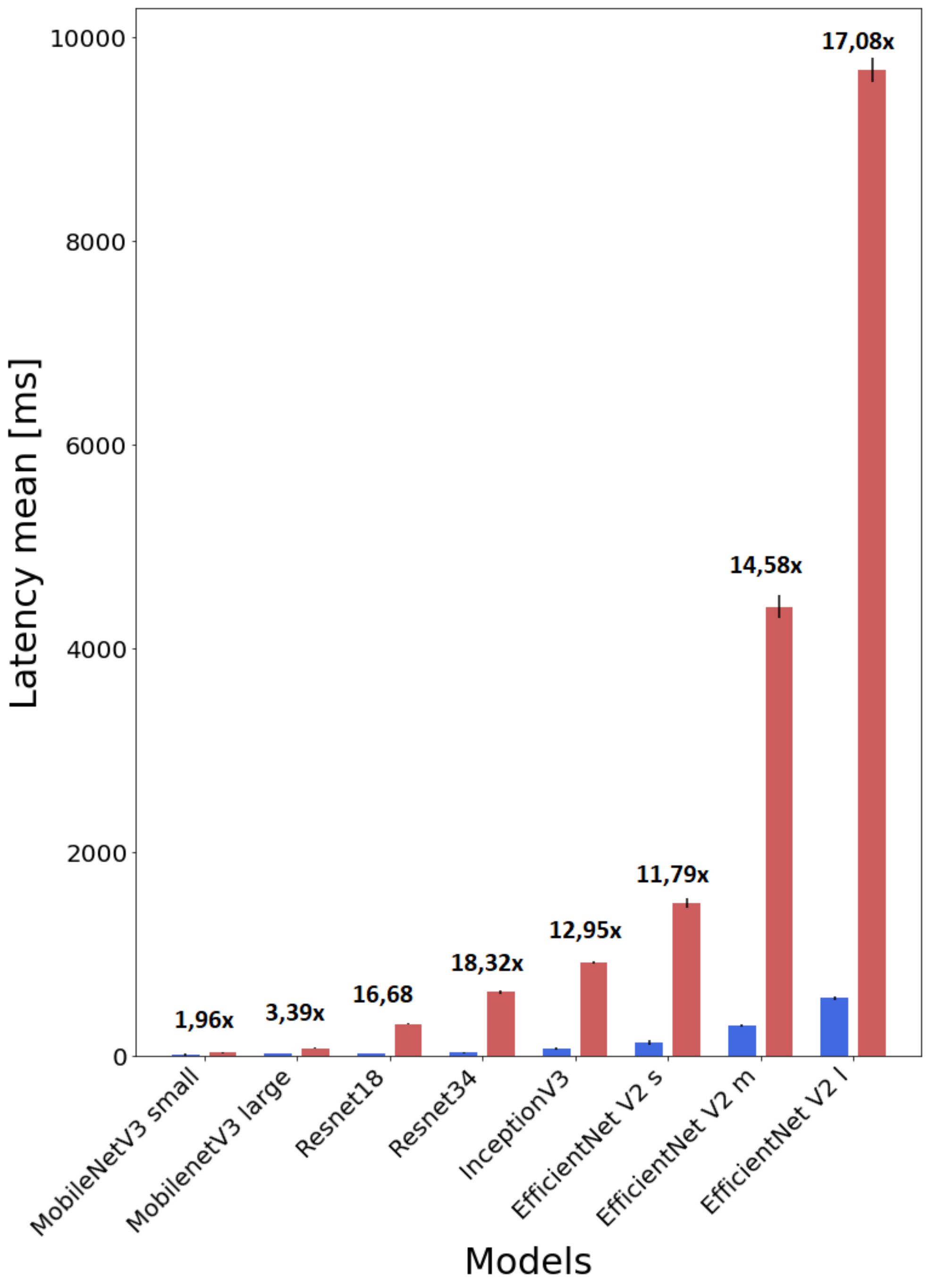


Figure 5.1: Comparison between latency mean of Pytorch and ONNX models. Models are tested on Device 1 (table 4.3) for this comparison.

sometimes is prohibitive.
In conclusion, until now, ONNX models work very well and their performances are comparable with Pytorch ones only if they are small or optimized for all types of hardware.

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge computing: Vision and challenges». In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on pp. 2, 14).

[2] *WebAssembly*. URL: `https://webassembly.org/`. (accessed: 03.02.2023) (cit. on pp. 2, 29).

[3] Yun Ma, 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*. 2019, pp. 1234–1244 (cit. on p. 3).

[4] *ONNX*. URL: `https://onnx.ai/index.html`. (accessed: 19.01.2023) (cit. on pp. 3, 25).

[5] ONNX Runtime developers. *ONNX Runtime*. `https://onnxruntime.ai/`. Version: x.y.z. 2021 (cit. on p. 3).

[6] *Pytorch*. URL: `https://pytorch.org/get-started/locally/`. (accessed: 19.01.2023) (cit. on pp. 3, 42).

[7] Warren S McCulloch and Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133 (cit. on p. 5).

[8] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 5).

[9] *Architecture of Perceptron*. URL: `https://ai.plainenglish.io/the-rise-and-fall-of-the-perceptron-c04ae53ea465`. (accessed: 08.02.2023) (cit. on p. 6).

[10] Sung Eun Kim and Il Won Seo. «Artificial Neural Network ensemble modeling with conjunctive data clustering for water quality prediction in rivers». In: *Journal of Hydro-environment Research* 9 (Apr. 2015). DOI: `10.1016/j.jher.2014.09.006` (cit. on p. 9).

[11] *Receiver operating characteristic.* URL: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. (accessed: 28.01.2023) (cit. on p. 11).

[12] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* URL: `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`. (accessed: 28.01.2023) (cit. on p. 12).

[13] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. «Edge intelligence: The confluence of edge computing and artificial intelligence». In: *IEEE Internet of Things Journal* 7.8 (2020), pp. 7457–7469 (cit. on pp. 15, 22).

[14] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. «Bringing the web up to speed with WebAssembly». In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2017, pp. 185–200 (cit. on p. 19).

[15] *Portability.* URL: `https://webassembly.org/docs/portability/`. (accessed: 08.02.2023) (cit. on p. 19).

[16] Khronos. *WebGL.* URL: `https://www.khronos.org/webgl/`. (accessed: 05.10.2022) (cit. on pp. 19–21).

[17] *AssemblyScript.* URL: `https://www.assemblyscript.org/`. (accessed: 08.02.2023) (cit. on p. 20).

[18] *WebGPU.* URL: `https://www.w3.org/TR/webgpu/`. (accessed: 05.10.2022) (cit. on p. 21).

[19] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. «MLitB: machine learning in the browser». In: *PeerJ Computer Science* 1 (2015), e11 (cit. on p. 21).

[20] *Tensorflow Playground.* URL: `https://playground.tensorflow.org/`. (accessed: 04.02.2023) (cit. on p. 21).

[21] *synaptic.* URL: `https://caza.la/synaptic/#/`. (accessed: 05.10.2022) (cit. on p. 21).

[22] *ConvNetJS.* URL: `https://cs.stanford.edu/people/karpathy/convnetjs/`. (accessed: 05.10.2022) (cit. on p. 21).

[23] *Keras.js.* URL: `https://github.com/transcranial/keras-js`. (accessed: 05.10.2022) (cit. on p. 21).

[24] *WebDNN.* URL: `https://mil-tokyo.github.io/webdnn/`. (accessed: 05.10.2022) (cit. on p. 21).

[25] *Tensorflow.js*. URL: `https://www.tensorflow.org/js`. (accessed: 05.10.2022) (cit. on p. 21).

[26] *Teachable Machine*. URL: `https://teachablemachine.withgoogle.com/`. (accessed: 04.02.2023) (cit. on p. 22).

[27] *MorphCast*. URL: `https://www.morphcast.com/`. (accessed: 04.02.2023) (cit. on p. 22).

[28] Brian J Spiesman, Claudio Gratton, Richard G Hatfield, William H Hsu, Sarina Jepsen, Brian McCornack, Krushi Patel, and Guanghui Wang. «Assessing the potential for deep learning and computer vision to identify bumble bee species from images». In: *Scientific reports* 11.1 (2021), pp. 1–10 (cit. on p. 22).

[29] Zekun Wang, Pengwei Wang, Peter C Louis, Lee E Wheless, and Yuankai Huo. «Wearmask: Fast in-browser face mask detection with serverless edge computing for covid-19». In: *arXiv preprint arXiv:2101.00784* (2021) (cit. on p. 22).

[30] Sangwoo Ryu, Kyungchan Ko, and James Won-Ki Hong. «Performance Analysis of Applying Deep Learning for Virtual Background of WebRTC-based Video Conferencing System». In: *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE. 2021, pp. 53–56 (cit. on p. 22).

[31] Vlad Pandelea, Edoardo Ragusa, Tommaso Apicella, Paolo Gastaldo, and Erik Cambria. «Emotion Recognition on Edge Devices: Training and Deployment». In: *Sensors* 21.13 (2021), p. 4496 (cit. on p. 22).

[32] Pei Ren, Xiuquan Qiao, Yakun Huang, Ling Liu, Calton Pu, and Schahram Dustdar. «Fine-grained elastic partitioning for distributed dnn towards mobile web ar services in the 5g era». In: *IEEE Transactions on Services Computing* (2021) (cit. on p. 22).

[33] Albin Correya, Pablo Alonso-Jiménez, Jorge Marcos-Fernández, Xavier Serra, and Dmitry Bogdanov. «Essentia TensorFlow models for audio and music processing on the web». In: *Web Audio Conference (WAC 2021)*. 2021 (cit. on p. 22).

[34] Jing Zhang and Dacheng Tao. «Empowering things with intelligence: a survey of the progress, challenges, and opportunities in artificial intelligence of things». In: *IEEE Internet of Things Journal* 8.10 (2020), pp. 7789–7817 (cit. on p. 22).

[35] Steven Euijong Whang, Yuji Roh, Hwanjun Song, and Jae-Gil Lee. «Data Collection and Quality Challenges in Deep Learning: A Data-Centric AI Perspective». In: *arXiv preprint arXiv:2112.06409* (2021) (cit. on p. 23).

[36] *Visual Studio Code.* URL: `https://code.visualstudio.com/`. (accessed: 19.01.2023) (cit. on p. 24).

[37] *Netron.* URL: `https://github.com/lutzroeder/netron`. (accessed: 10.10.2022) (cit. on p. 26).

[38] *ONNX concept.* URL: `https://onnx.ai/onnx/intro/concepts.html`. (accessed: 19.01.2023) (cit. on p. 26).

[39] *ONNX Model Zoo.* URL: `https://github.com/onnx/models`. (accessed: 10.10.2022) (cit. on p. 26).

[40] OpenJS Foundation. *Node.js.* URL: `https://Node.js`. (accessed: 16.01.2023) (cit. on p. 28).

[41] Vercel. *Next.js.* URL: `https://Next.js`. (accessed: 16.01.2023) (cit. on p. 28).

[42] Meta Platforms. *React.js.* URL: `https://React.js`. (accessed: 16.01.2023) (cit. on p. 28).

[43] *webpack.* URL: `https://webpack.js.org`. (accessed: 16.01.2023) (cit. on p. 28).

[44] *ONNX Runtime Architecture.* URL: `https://onnxruntime.ai/docs/reference/high-level-design.html`. (accessed: 24.01.2023) (cit. on p. 30).

[45] Kate Saenko, Brian Kulis, Mario Fritz, and Trevor Darrell. «Adapting visual category models to new domains». In: *European conference on computer vision.* Springer. 2010, pp. 213–226 (cit. on p. 31).

[46] Abhay Gupta, Arjun D'Cunha, Kamal Awasthi, and Vineeth Balasubramanian. «Daisee: Towards user engagement recognition in the wild». In: *arXiv preprint arXiv:1609.01885* (2016) (cit. on p. 31).

[47] Andrew Howard et al. «Searching for mobilenetv3». In: *Proceedings of the IEEE/CVF international conference on computer vision.* 2019, pp. 1314–1324 (cit. on p. 33).

[48] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. «Netadapt: Platform-aware neural network adaptation for mobile applications». In: *Proceedings of the European Conference on Computer Vision (ECCV).* 2018, pp. 285–300 (cit. on p. 34).

[49] Barret Zoph and Quoc V Le. «Neural architecture search with reinforcement learning». In: *arXiv preprint arXiv:1611.01578* (2016) (cit. on pp. 34, 35).

[50] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «Mobilenets: Efficient convolutional neural networks for mobile vision applications». In: *arXiv preprint arXiv:1704.04861* (2017) (cit. on p. 34).

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. «Mobilenetv2: Inverted residuals and linear bottlenecks». In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, pp. 4510–4520 (cit. on p. 34).

[52] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. «Mnasnet: Platform-aware neural architecture search for mobile». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2019, pp. 2820–2828 (cit. on p. 34).

[53] Kaiming He, 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.* 2016, pp. 770–778 (cit. on p. 34).

[54] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. «Rethinking the inception architecture for computer vision». In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 2818–2826 (cit. on p. 35).

[55] Mingxing Tan and Quoc Le. «Efficientnetv2: Smaller models and faster training». In: *International Conference on Machine Learning.* PMLR. 2021, pp. 10096–10106 (cit. on p. 35).

[56] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «ImageNet: A large-scale hierarchical image database». In: *2009 IEEE Conference on Computer Vision and Pattern Recognition.* 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848` (cit. on p. 37).

[57] Google Tesla Priceton University ETHzurich Equinix. *Activeloop.* URL: `https://datasets.activeloop.ai/docs/ml/datasets/daisee-dataset/`. (accessed: 11.01.2023) (cit. on p. 37).

[58] Mozilla Foundation. *Navigator.deviceMemory documentation.* URL: `https://developer.mozilla.org/en-US/docs/Web/API/Navigator/deviceMemory`. (accessed: 13.02.2023) (cit. on p. 47).