VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
INFORMATIKOS KATEDRA

Magistro baigiamasis darbas

Mikrovaldikliams skirtų kompiuterinės regos metodų kūrimo tyrimai

Atliko:
Volodymyr Kadzhaia (parašas)

Darbo vadovas:
Prof. Dr. Olga Kurasova (parašas)

Recenzentas:
Prof. Dr. Saulius Gražulis (parašas)

Vilnius
2023

Contents

Introduction

A microcontroller is a computer on a single integrated circuit that includes a CPU, RAM, some form of ROM, and I/O ports. It has a great impact on our lives, which cannot be ignored. Unlike a general-purpose computer, microcontrollers are dedicated to performing a specified task and executing a single application. Automatically controlled products like automatic engine control systems, remote controls, power tools, toys, and office machines i.e., photo-copier, printers, and fax machines which are used commonly, are being programmed using microcontrollers [HHH16].

Recently, more and more smart objects [Bor14] have appeared, such as smartphones, smartwatches, and even smart bulbs. All these things simplify our life, for example, modern cars can make a sound in the cabin or stop when an object is detected. In order to implement a smart device, scientists decided to turn on artificial intelligence. Artificial intelligence, using a variety of algorithms, allows you to simulate human consciousness, so when using algorithms, one can implement object detection. This is how one of the most popular libraries among microcontrollers – TinyML [Ray20], was implemented. This library was implemented on the basis of the existing and popular open-source Tensorflow library. The neural network is first trained using a large amount of pre-collected data on a powerful machine and then transferred to microcontrollers. This leads to a static model that is difficult to adapt to new data and impossible to adapt to different scenarios, which reduces the agility of the Internet of Things (IoT). This is how TinyML works. There are several approaches to creating flexibility, one of the methods is using TinyOL (TinyML with Online-Learning [RAR21]). This approach allows you to add streaming training [RAR21] to the device. TinyOL [RAR21] is based on the concept of online learning and is suitable for limited IoT devices [RAR21]. Thus, TinyML can be used in Arduino to develop a thermodynamic model [CDM19] or for training. In addition, now more and more specific microcontrollers appear that are designed to work with AI, for example, Arduino 33 BLE Sense, as well as libraries specifically for these microcontrollers. Thus, large companies have to create their own library or implement a framework based on an existing library. One important component, even in the development of such libraries is the software architecture. It is important to understand what processes will be involved, how the components (classes and data structures) will be connected, and how the program is optimized. Therefore, programmers are trying to develop artificial intelligence (AI), which is used to develop complex algorithms for protecting networks and systems, including IoT systems. But such systems have their drawbacks, one of them being the processing of large data since the microcontroller has a limited amount of memory and also has a single-threaded channel (one clock per cycle). For example, computer vision requires a huge amount of calculations; for this, a cloud and special libraries were developed that perform calculations locally on more powerful systems, after which the results are entered into a special file readable for microcontrollers. There are many examples in the literature of simple computer vision algorithms proving to be extremely useful in a variety of applications [BBV00; Hor93; LBG97; NAT+97; RRN02; SBW+97; UN00].

The goal of this work is to compare classical methods and deep learning methods in developing computer vision for microcontrollers.

Achievement of the goal is carried out with the help of the tasks set:

- to analyse IoT technologies as well as classical machine learning and deep learning techniques and to identify methods suitable for developing computer vision for microcontrollers;
- to find several datasets that can be used to conduct a comparative analysis of the methods;
- to develop an application using different third-party libraries, such as Tensorflow and OpenCV, to determine the feasibility of training machine learning and deep learning models on a microcontroller;
- to conduct a comparative analysis of classical and deep learning methods through experimental investigations.

1 Literature Review

Internet of Things (IoT) devices typically operate on microcontrollers, which are relatively simple computer chips devoid of an operating system. These microcontrollers have minimal processing power and considerably less memory than a standard smartphone, a factor that poses challenges in running local pattern-recognition tasks such as deep learning on IoT devices [Ack20].

Despite these constraints, advancements in machine learning and hardware design have provided innovative solutions. For instance, researchers have developed systems like MCUNet, a technology that designs compact neural networks capable of delivering exceptional speed and accuracy for deep learning on IoT devices. Remarkably, this high level of performance is achieved despite the limited memory and processing power inherent in these devices [Ack20].

The development and integration of such systems not only facilitates the expansion of the IoT universe but also leads to significant energy savings and enhancements in data security. The ability to perform complex computations locally also reduces the need for constant communication with a cloud server, thereby further increasing efficiency and reducing potential points of vulnerability [Ack20].

Moreover, the advancements in machine learning have created opportunities to integrate a level of intelligence in low-end IoT nodes, such as microcontrollers. Various surveys have evaluated different machine learning methods that can address the unique challenges presented by IoT data [MRB+18]. This integration of machine learning with IoT devices and microcontrollers points towards a future where devices not only connect and communicate but also learn and adapt independently to the changing environment.

1.1 IoT

The Internet of Things (IoT) comprises physical objects connected to the internet, enabling data exchange. These objects include wearable technology, smart home devices, and industrial equip-

ment in smart factories [PP16]. Kevin Ashton coined the IoT concept in 1999 while developing Radio Frequency Identification (RFID) technology, which allows objects to connect to the internet [Lan05].

RFID technology's advantages include low power consumption, low cost, and a small client form factor [MRV+17]. Furthermore, advancements in sensor technology have accelerated IoT's implementation, fostering digital transformation in numerous industries.

Microelectromechanical systems (MEMS), miniature devices created from mechanical and electrical components, have emerged due to technological improvements. These sensors, crucial to the sensor industry, offer high performance, low power consumption, and low manufacturing cost. They are increasingly used in IoT systems for real-time data collection and analysis, enhancing decision-making and automation across various sectors.

It's important to note that while MEMS is widely used, other technologies like Nano-electromechanical systems (NEMS) and bio-sensors also exist. However, MEMS's scalability and cost-effectiveness make it a popular choice, enabling sensor miniaturization for various applications [DT18].

1.1.1 IoT architecture

IoT architecture consists of different layers of technologies supporting IoT (Fig. 1). It serves to illustrate how various technologies relate to each other and to communicate the scalability, modularity, and configuration of IoT deployments in different scenarios [PP16]. To promote the deployment of IoT systems, the architecture must be scalable to manage the increasing number of devices and services without compromising their performance. It must be interoperable, allowing devices from different vendors to cooperate to achieve common goals. In addition, the architecture must be distributive to enable the creation of a distributed environment where data is collected from various sources and processed by different entities in a distributed manner. It should be able to operate with minimal resources since IoT objects generally have limited computing power. Lastly, the architecture must be secure to prevent unauthorized access to sensitive data. Currently, there is no single reference architecture, and creating one is proving very complicated despite many standardization efforts. The main problem lies in the natural fragmentation of possible applications, each of which depends on many very often different variables and design specifications [LPS21]. This problem must be added to each supplier's tendency to propose its platform for similar applications [LPS21].

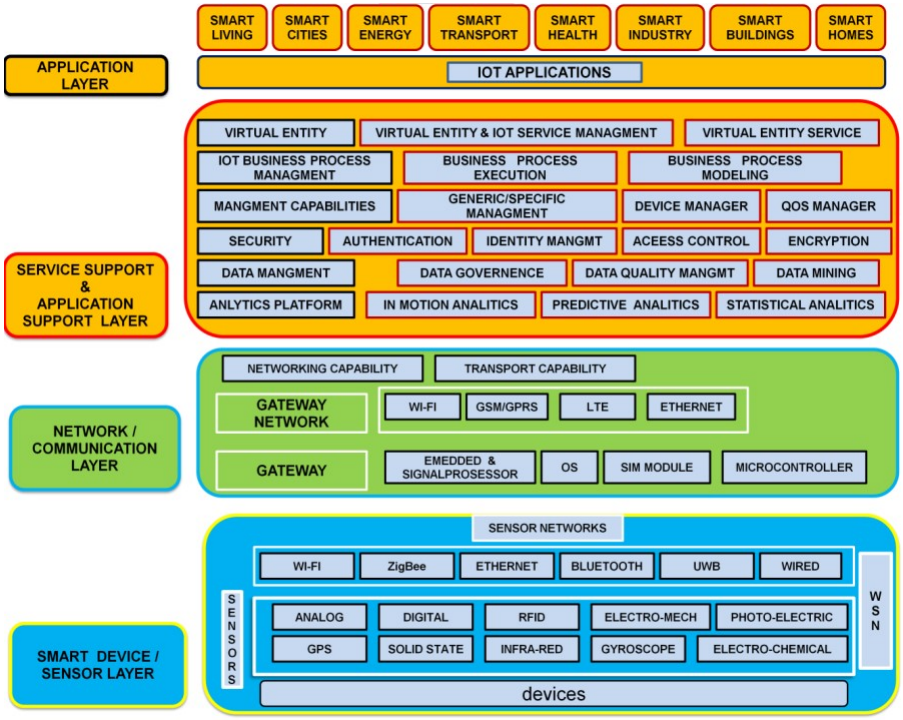


Figure 1. IoT architecture (image is under Creative Commons Attribution License) [PP16]

1.1.1.1. End Devices

Devices are the objects that actually make up the ”Things” in the Internet of Things. They act as an interface between the real and digital worlds and take on different sizes, shapes, and levels of technological sophistication depending on the task they perform within a particular IoT deployment.

1.1.1.2. Software

The software is responsible for communicating with the cloud, data collection, device integration, and real-time data analysis. It also provides opportunities for data visualization and interaction with the IoT system.

1.1.1.3. Communication (Protocols)

The network layer has the task of transporting the data provided by the perception level to the application layer. It includes all the technologies and protocols that make this connection possible and should not be confused with the network layer of the ISO/OSI model, which only routes data within the network along the best path. There are a large number of protocols that can be used in IoT [LPS21].

1.1.1.4. Platform

An IoT platform is a place where all data is collected, analyzed, and transmitted to the user in a convenient form. Platforms can be installed locally or in the cloud. The choice of platform depends

on the requirements of a particular IoT project and many factors: architecture and technology stack, reliability, settings, protocols used, hardware independence, security, efficiency, and cost.

1.1.1.5. Secure

With the advent of IoT and its use in all areas of activity, the question of security has arisen. IoT systems carry significant business value, and smart objects also become vulnerable to cybercrime, resulting in data leakage, including confidential information. Therefore, many developers and scientists have thought about protecting endpoints, and networks and moving data over them, which means creating a scalable security paradigm [ABJ19].

1.1.2 Vision sensors in IoT

In IoT, automation is provided by passing data to a device. Sensors and actuators in the IoT represent these two endpoints of the system.

A sensor is generally a device capable of detecting changes in an environment. A sensor is able to measure a physical phenomenon and transform it into an electric signal [ABJ19].

There are several types of sensors in IoT, such as temperature sensors, humidity, pressure, proximity, optical, chemical, etc.

1.1.2.1. Optical sensors

Optic sensing technology is used to detect electromagnetic energies such as light. It uses the photoelectric effect concept, which says electrons will be ejected when a negatively charged plate of some suitable light-sensitive material is hit by a photon beam [ABJ19]. Optical sensors are devices that use light to detect and measure the physical properties of objects or environments. There are many types of optical sensors, including photoelectric sensors, photodiodes, phototransistors, and charge-coupled devices (CCDs).

Photoelectric sensors use a light-sensitive element, such as a photodiode or phototransistor, to detect the presence or absence of an object. They are commonly used in applications such as automated door openers, object counting, and material handling. Photodiodes are semiconductor devices that convert light into an electrical current. They are used in a variety of applications, including solar cell panels, barcode scanners, and medical imaging. Phototransistors are transistors that use light to control the flow of current. They are commonly used in proximity sensors, light-activated switches, and light-sensitive alarm systems.

Optical sensors have a number of advantages (for example, in telecommunications, optical sensors play a pivotal role due to their capability to operate over large distances and their immunity to electromagnetic interference) over other types of sensors. They are sensitive to a wide range of wavelengths, they can operate over large distances, and they are immune to electromagnetic interference. However, they can be affected by ambient light, and they may not be as accurate as some other types of sensors in certain applications.

1.1.2.2. Vision sensors

By applying image processing to images captured by a camera, the vision sensor calculates the characteristics of an object, such as its area, center of gravity, length, or position, and outputs the data or judgment results. The machine vision system converts the taken target into an image signal through the machine vision product. The image signal is then transmitted to a dedicated image processing system. According to information such as pixel distribution, brightness, and color, the image signal is converted to a digitized signal. The imaging system performs various operations to extract the features of the target, and in turn, according to the result of the discrimination, the image is controlled.

Vision sensors use cameras and image processing algorithms to analyze visual information and make decisions based on that information. These sensors can be used for a wide range of applications, including object recognition, tracking, and measurement. The process of using a vision sensor begins with capturing an image of the target object using a camera. The image is then processed by an image processing algorithm, which calculates various characteristics of the object, such as its area, center of gravity, length, or position. This processed data is then outputted as a digital signal, which can be used for further analysis or control.

One of the key benefits of using vision sensors is their ability to analyze visual information in real time, which enables faster and more accurate decision-making. Additionally, vision sensors can be used in environments where other types of sensors may not be suitable, such as in low-light conditions or for monitoring large areas. It is worth mentioning that vision sensors can be integrated with other types of sensors to provide more accurate and robust data. For example, vision sensors can be used in conjunction with other sensors, such as infrared, ultrasonic, or LIDAR to provide a more comprehensive understanding of the environment.

1.1.2.3. Fiber optic sensors

Fiber optic sensors (FOS) are devices designed to record changes in system performance and broadcast a signal over a fiber optic channel. Such sensors can be used to monitor temperature and mechanical stress, they are also used to control pressure, vibration, and other indicators. They are used in the construction industry, utilities, mining, etc.

This type of sensor is based on optical fiber. It is a core in a polymer shell, through which the light flux passes. The core is made of glass or plastic, which is supplied with special additives to improve the refractive index of light waves.

Fiber optic sensors use an optical fiber as a signal transmission line or sensing element. Sensors with optical converters have gained the greatest demand. Such a system consists of a sensitive optical element, a receiver, and an emitter. The converter is placed between the end parts of the receiving and transmitting fibers, and the LED can play the role of the emitter.

1.1.3 Image Sensors

The solid-state image sensor is a critical component of photo-electronic devices such as mobile phones, digital video cameras, automotive imaging, surveillance, biometrics, etc. [YYH18]. Two types of solid-state image sensor technologies have been developed: Charged Coupled Devices (CCD) and CMOS Image Sensors (CIS) [YYH18]. CCD image sensor technology [HM78; Hol96] has been the dominant electronic imaging technology since the 1970s [YYH18]. A CCD sensor is composed of a photodetector and a series of metal oxide semiconductor (MOS) capacitors [YYH18]. The charge generated by the photosensitive detector is transferred out through the capacitors [YYH18]. A special manufacturing process is needed to create the ability to transport charge across the chip without distortion [YYH18]. This special process leads to high-quality sensors in terms of fidelity and light sensitivity, but the cost is high [YYH18].

The main difference between an image sensor and a vision sensor lies in their respective functionalities and applications:

- Image Sensor: An image sensor is primarily responsible for capturing visual data in the form of images or video. It detects light intensity and color information, converting them into electrical signals that can be processed and stored. Image sensors are commonly used in digital cameras, smartphones, surveillance systems, and many other devices that require visual information capture.

- Vision Sensor: A vision sensor, on the other hand, encompasses a broader scope. It refers to a sensor or a combination of sensors that not only capture images but also perform advanced processing and analysis tasks on the visual data. Vision sensors often incorporate image processing algorithms and artificial intelligence techniques to extract useful information from the captured images. They can perform tasks like object recognition, tracking, depth estimation, and scene analysis. Vision sensors find applications in robotics, autonomous vehicles, industrial automation, and various other fields where visual perception and understanding are required.

1.1.3.1. CMOS image sensors

The overall architecture of a CMOS image sensor is shown in Fig. 2.

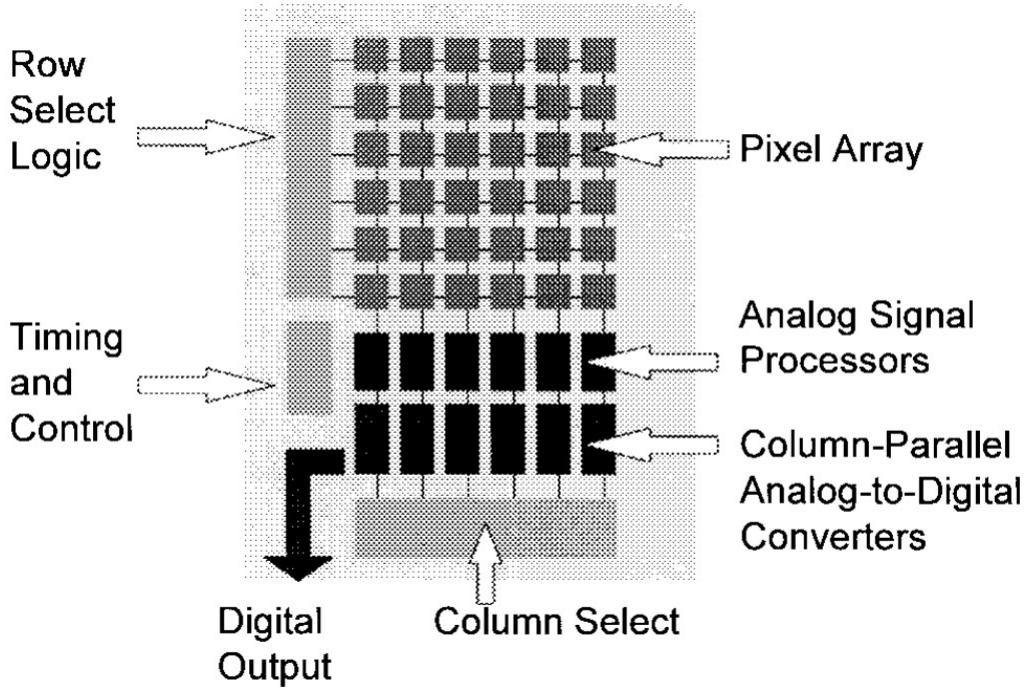


Figure 2. The architecture of a CMOS image [Fos97]

The image sensor consists of an array of pixels that are typically selected a row at a time by a row select logic [Fos97]. CCD (Charge-Coupled Device) and CMOS (Complementary Metal-Oxide-Semiconductor) image sensors have an array of pixels. The main difference between the two types of sensors lies in the way they capture and read out the image data. In a CCD sensor, the charge collected by each pixel is transferred through a limited number of output nodes to be converted to voltage, buffered, and sent off-chip as an analog signal. In contrast, in a CMOS sensor, each pixel has its own charge-to-voltage conversion, and the sensor often includes amplifiers, noise-correction, and digitization circuits, so that the chip outputs digital bits. This fundamental difference leads to several other differences in terms of power consumption, speed, noise performance, etc. This can be either a shift register or a decoder [Fos97]. In the context of an image sensor, both a shift register and a decoder can be used to select a row of pixels for readout. A shift register is a type of sequential logic circuit that can store and shift data. In this case, it can be used to store and shift a binary value representing the selected row. A decoder, on the other hand, is a combinational logic circuit that converts an n-bit binary input into 2^n outputs, with only one output being active at a time. In this case, it can be used to convert the binary representation of the selected row into an active output that selects the corresponding row of pixels. Both methods achieve the same result of selecting a row of pixels for readout but use different types of logic circuits to do so. The pixels are read out to vertical column busses that connect the selected row of pixels to a bank of analog signal processors (ASPs) [Fos97]. The image sensor has a system for selecting and reading out the data from its pixels. The data from the pixels is processed by analog signal processors (ASPs) and then converted to digital form by analog-to-digital converters (ADCs). The digital data is then selected for output by a column-select logic circuit, which can be either a shift register or a decoder. This circuit determines which column of data is read out from the sensor.

1.1.3.2. CCD image sensors

In a CCD sensor, the light (charge) incident on the sensor pixel is transmitted from the chip through a single output node, or through just a few output nodes. The charges are converted to a voltage level, accumulated, and sent out as an analog signal. This signal is then summed and converted to numbers by an analog-to-digital converter outside the sensor.

The disadvantages of CCD sensors are that they are analog components, require more electronics "near" the sensor, are more expensive to manufacture, and can consume up to 100 times more power than CMOS sensors. CMOS sensors, on the other hand, are digital components that are less expensive to manufacture and consume less power than CCD sensors. They use a different technology for capturing digital images and can include additional processes such as amplification and noise correction[https://www.camerasource.com/industry-news/ccd-vs-cmos-sensor-image-quality/]. Increased power consumption can also increase the temperature in the camera itself, which not only affects image quality and increases the cost of the final product, but also the environmental impact. CCD sensors also require faster data transfer because all data passes through just one or a few output amplifiers. CCD sensors have a limited number of channels (usually two) where the pixel data is transferred off the chip [Dat16]. This means that all data must pass through just one or a few output amplifiers, which can require faster data transfer to process the image efficiently.

In contrast, CMOS sensors can have an arbitrary number of channels and some sensors have as many as sixteen [Dat16].

1.1.4 Vision Sensors vs. Vision Systems

In some cases, vision sensors and machine vision systems may both be able to satisfy an operation's needs. Different models are designed to meet varying price and performance requirements. Vision sensors are similar to machine vision systems in their powerful vision algorithms, self-contained and industrial-grade hardware, and high-speed image acquisition and processing. In the context of machine vision, "powerful" typically refers to the ability of vision algorithms to accurately and reliably perform complex image analysis tasks. The power of vision algorithms can be measured in several ways, including their accuracy, speed, and robustness. Accuracy refers to the ability of the algorithm to correctly identify and analyze the features of interest in an image. Speed refers to the time it takes for the algorithm to process an image and produce a result. Robustness refers to the ability of the algorithm to perform well under varying conditions, such as changes in lighting or object orientation. They are both designed to perform highly-detailed tasks on high-speed production lines. While machine vision systems perform guidance and alignment, optical character recognition, code reading, gauging, and metrology, vision sensors are purpose-built to determine the presence/absence of parts and generate simple pass/fail results [Dat20].

1.2 Machine learning methods and IoT

Machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence (AI). The changes might be either enhancement to already performing systems [Nil93]. These changes can include increased accuracy, efficiency, and adaptability. Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. [Nil93].

With the advent of IoT devices, devices have become smarter than previous generations of devices and can communicate with each other, transfer data, or collect data. In many IoT applications, devices can be programmable, perform predetermined actions based on some predefined conditions, or give feedback from collected data. IoT devices need not only to collect data, and communicate with other devices, but also to be autonomous [ZKK+19]. These conditions can vary depending on the specific application and the desired behavior of the device. For example, a smart thermostat might be programmed to turn on the heating system when the temperature in a room falls below a certain threshold. The specific conditions that are used will depend on the requirements of the application and the desired behavior of the device. To do this, devices must make decisions and learn independently from the data they collect. An autonomous device is one that can operate independently without the need for human intervention. In order to do this, the device must be able to make decisions based on the data it collects. Machine learning algorithms can enable a device to learn from the data it collects and improve its decision-making over time. By learning from the data, the device can adapt to changing conditions and make more accurate predictions or decisions. This can improve the performance of the device and enable it to operate more effectively in an autonomous manner. Therefore, the development or deployment of machine learning algorithms in IoT can significantly improve the quality of applications as well as the infrastructure itself [AAB+20].

1.2.1 Definition of classical methods and deep learning methods

Classical methods, also known as traditional machine learning methods, are algorithms that are based on mathematical models and statistical approaches. These methods involve the use of handcrafted features, which are input variables that are chosen by the data scientist to represent the data. Some examples of classical machine learning methods include linear regression, logistic regression, decision trees, and support vector machines (SVMs).

Deep learning methods, on the other hand, are a type of artificial neural network composed of many layers. These layers are able to learn and extract features from the data automatically without the need for human input. Deep learning models are able to learn patterns and relationships in data that are too complex or too numerous for humans to identify. Some examples of deep learning methods include convolutional neural networks (CNNs), recurrent neural networks (RNNs), and autoencoders [BKG20].

1.2.2 Machine Learning tasks

Machine learning tasks are the types of prediction or inference being made, based on the problem or question that is being asked, and the available data. Some common machine learning tasks include

binary classification, multiclass classification, regression, clustering, anomaly detection, ranking, recommendation, forecasting, image classification and object detection.

In machine learning exists various tasks such as:

1. Classification: Classification is a supervised learning task where a model is trained on a dataset with pre-labeled examples, and the task is to assign new, unseen examples to one of a finite number of discrete categories, or ”classes”. Examples of this might be email spam detection (where the classes are ”spam” and ”not spam”) or digit recognition (where the classes are the digits from 0 to 9).

2. Clustering: Clustering is an unsupervised learning task that involves grouping a set of data points in such a way that data points in the same group (called a cluster) are more similar to each other than to those in other groups. Clustering is often used when you don't have labeled data and want to identify patterns or structure in your dataset.

3. Dimensionality Reduction: Dimensionality reduction is a technique that reduces the number of input variables in a dataset. It's often used when dealing with high-dimensional data, to help visualize the data and to improve computational efficiency. Principal Component Analysis (PCA) and t-distributed Stochastic Neighbor Embedding (t-SNE) are popular methods for dimensionality reduction.

4. Anomaly Detection: Anomaly detection is the identification of rare items, events, or observations which raise suspicions by differing significantly from the majority of the data. These anomalies could represent things like bank fraud, medical problems, or errors in a text. Anomaly detection is often used when the number of anomalies in the data is very low, and it's hard to obtain a robust set of anomaly examples to use for training in a supervised learning context.

1.2.3 Machine Learning types

1.2.3.1. Supervised Learning

Supervised learning is a fundamental aspect of machine learning where an algorithm learns from labeled training data, and then applies what it has learned to new, unseen data [BJT+10; WZZ+21]. In supervised learning, each instance in the training set consists of input features and an expected output, also known as the label or class. The relationship between these features and the class is initially unknown. The main objective of supervised learning is to establish a model that can accurately map the input features to the class, thereby uncovering the unknown relationship between them [WZZ+21]. Supervised learning can be divided into two primary tasks based on the type of output variable: classification and regression. Classification tasks predict categorical output variables, while regression tasks predict continuous output variables [WZZ+21]. An example of a classification task might be the identification of fraudulent transactions, where the possible classes are ”fraudulent” or ”not fraudulent.” An example of a regression task might be predicting house

prices based on features like the number of bedrooms, location, and age of the house. The effectiveness of a supervised learning model relies heavily on the quality and quantity of the training data. For high-dimensional data, traditional methods might struggle, particularly when the sample size is small. These traditional methods typically perform well when there is a large sample size [PCC16]. Supervised learning has a wide range of applications, including computer vision and image understanding, data mining and knowledge discovery, and natural language and document processing [PCC16].

The performance of a supervised learning model is measured using various metrics such as accuracy, precision, recall, F1 score, and mean squared error, which depend on whether the task is classification or regression.

1.2.3.2. Unsupervised Learning

Unsupervised learning [TAH06] is one of the branches of machine learning. It studies a wide class of data processing problems in which only descriptions of a set of objects (training sample) are known, and it is required to detect internal relationships, dependencies, and patterns that exist between objects. The term "wide class" in this context is a somewhat subjective and general term used to denote the expansive and diverse range of data types and structures that unsupervised learning can handle. Unsupervised learning is an approach in machine learning where the algorithm learns from the input data without any explicit output labels or answers provided. This contrasts with supervised learning, which relies on labeled examples during the learning process.

The diversity or 'width' of data in unsupervised learning is characterized by a few aspects:

- Data types: Unsupervised learning can handle a wide array of data types, from numeric (continuous or discrete) and categorical data, to more complex types like text, images, audio, and even mixed data types.
- Data structures: Unsupervised learning algorithms can process structured data (like database data), semi-structured data (like XML or JSON data), and unstructured data (like raw text or images).
- Domains: Unsupervised learning is domain-agnostic and can be applied across a multitude of fields, such as marketing (for customer segmentation), biology (for gene clustering), finance (for anomaly detection in credit card transactions), and many more.
- Data dimensionality: From low-dimensional data to high-dimensional data, unsupervised learning techniques can be used. In some cases, dimensionality reduction techniques (which are a part of unsupervised learning) are specifically used for dealing with very high-dimensional data.
- Size of the dataset: Unsupervised learning can handle small to very large datasets. Particularly, when labeled data is scarce, unsupervised learning can be a useful approach as it doesn't require output labels.

Unlike supervised learning, unsupervised learning methods typically aren't used for regression or classification problems, as these methods operate without predefined output labels [BS20]. Common algorithms used in unsupervised learning include [BS20]:

- clustering,
- anomaly detection,
- neural networks,
- approaches for learning latent variable models.

1.2.3.3. Semi-supervised Learning

Semi-supervised learning is one of the machine learning methods that use both labeled and unlabeled data during training. Typically, a small amount of labeled and a significant amount of unlabeled data is used. Semi-supervised learning is a compromise between unsupervised learning (without any labeled training data) and supervised learning (with a fully labeled training set).

The ultimate goal of a semi-supervised learning model is to provide a better outcome for prediction than that produced using the labeled data alone from the model [Sar21].

1.2.3.4. Reinforcement Learning

Reinforcement learning is a type of machine learning algorithm that enables an agent to learn by interacting with its environment [Sar21]. The agent takes actions in the environment and receives feedback in the form of rewards or penalties. The goal of the agent is to learn a policy that maximizes the cumulative reward over time. The agent does this by exploring the environment and updating its policy based on the feedback it receives. Over time, the agent learns to take actions that lead to higher rewards and improve its efficiency in achieving its goal.

Reinforcement learning is different from supervised learning in that it does not rely on pre-labeled data to learn. Instead, an agent learns by interacting with its environment and receiving feedback in the form of rewards or penalties. The agent must balance the trade-off between exploration and exploitation. Exploration refers to the agent taking actions to gather new information about the environment, while exploitation refers to the agent using its existing knowledge to take actions that maximize its reward. The goal of reinforcement learning is to find an optimal policy that balances exploration and exploitation to maximize the cumulative reward over time.

1.2.4 Classical Machine Learning Algorithms in Computer Vision

1.2.4.1. k-Nearest Neighbors (k-NN)

K-Nearest Neighbors (K-NN) is a non-parametric, instance-based learning algorithm. Non-parametric refers to its flexibility in model structure determined by the dataset, while instance-based learning implies that all training data is used during the testing phase, which makes training faster but slows down the testing process and makes it more costly.

The K-NN algorithm stores all available data and classifies new observations based on a similarity measure. The process involves calculating the distance to each object in the training set, selecting the k objects closest in distance, and assigning the class that appears most frequently among these k neighbors.

In the context of computer vision, such as face recognition, K-NN can identify and classify new images based on their similarity to stored images. Features of the images, like distances between facial features or skin texture, are compared to those of the stored images to assign the class of the most similar existing images.

1.2.4.2. Support Vector Machines

Support vector machine is a machine learning algorithm that learns to classify data points using labeled training samples [SC08]. Support Vector Machine (SVM) and Artificial Neural Networks (ANN) are both supervised learning algorithms used for classification tasks, learning from labeled training samples. However, their methodologies differ.

SVM functions by identifying the hyperplane that maximizes the margin between two classes in the training data [SC08]. This margin is the distance between the hyperplane and the closest data points from each class, known as support vectors. The algorithm strives to maximize this distance, resulting in the best hyperplane. SVM employs a technique known as the kernel trick to transform data into a higher dimensional space, allowing for separation of classes via a linear hyperplane.

Conversely, ANN is a computational model inspired by biological neural networks. Comprising layers of interconnected nodes or neurons, it adjusts the weights of connections between neurons during training to minimize the error between predicted and true class labels.

The application of SVM in computer vision is one of the most successful applications of machine learning [ERF97] in this field. One of the most significant benefits of SVM in computer vision is its ability to handle high-dimensional data. Both SVM and ANNs are powerful machine learning algorithms that can effectively handle high-dimensional data. The benefit of using SVM in computer vision is its ability to handle high-dimensional data while maintaining good generalization performance. SVM does this by finding the optimal hyperplane that separates the data into different classes while maximizing the margin between the classes. This helps to prevent overfitting and improve the generalization performance of the algorithm. Computer vision applications often involve images, which can contain a large number of pixels, leading to a high-dimensional feature space. SVMs can effectively handle this high-dimensional data, making it an ideal algorithm for computer vision tasks.

An additional benefit of Support Vector Machines (SVM) in computer vision is its capability to learn complex, non-linear decision boundaries. This is critical in situations where the relationship between features and the target variable isn't linearly separable. SVM applies certain transformations to elevate the data into a higher-dimensional space. Within this space, it's possible to learn a linear decision boundary, thus facilitating the learning of non-linear decision boundaries. This linear boundary, referred to as a hyperplane, makes SVM effective in classifying complex patterns by maximizing the margin between classes.

Despite its benefits, SVM algorithms can be computationally expensive, and their performance may suffer when faced with a large number of training data. However, advances in computational resources and optimization techniques have helped address these issues, making SVM a popular algorithm for computer vision tasks.

1.2.4.3. Decision Tree

A decision tree is machine learning model, which is used for both classification and regression tasks. This model is visually represented in the form of a tree structure [RM05]. The decision tree begins with a node called the 'root' which has no incoming edges. From this root node, the tree expands with additional nodes based on certain conditions or rules.

In a standard decision tree learning algorithm, we start with a set of examples at the root node. A test or decision rule is applied to these examples, which splits them into two distinct subsets:

1. The first subset consists of examples that satisfy the established rule.
2. The second subset includes the examples that do not satisfy the established rule.

The process continues as each of these subsets is then subjected to a new decision rule. This results in more branches being formed in the tree, which in turn leads to the creation of more nodes. This procedure of applying decision rules and splitting subsets continues iteratively until a stopping condition is met [AKA02]. Stopping conditions could be reaching a certain tree depth, achieving a minimum node purity, or a minimum number of samples per leaf, among others.

The final nodes in the tree, which do not undergo any further splitting, are called leaf nodes' or simply leaves'. Each leaf node represents a decision outcome or a prediction, which is the output of the decision tree for a given input example [AKA02].

This structure of decision trees allows for intuitive graphical representations and easy interpretation of the decision-making process. However, decision trees can also suffer from overfitting, especially with complex trees, and may need pruning or other regularization methods to achieve the best performance.

1.2.4.4. Naive Bayes Algorithms

The Naive Bayes classifier is a machine learning algorithm designed for multi-class classification of data with independent features. In one pass, the conditional probability of each feature is calculated, then Bayes' theorem is applied to find the probability distribution of the observations.

Bayes' theorem is a simple mathematical formula used to calculate conditional probabilities. Classifier applies the Bayes theorem, presented in equation (1), for probabilistic classification [SNdA+22]. By observing the values (input data) of a given set of features or parameters, represented as B in the equation, the Naive Bayes classifier is able to calculate the probability of the input data belonging to a certain class, represented as A [SNdA+22].

$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$ (1)

In Bayesian methods, $P(A)$ is our initial guess about how likely it is that something will happen. This guess can be based on past data or what experts think. We use this guess along with new information to update our belief about how likely it is that something will happen. In the context of computer vision, Naive Bayes Classifiers are used for image classification, where the task is to assign a label to an input image, based on its features.

The "naive" in Naive Bayes classifiers refers to the assumption that the features in an image are conditionally independent of each other, given the class label. This assumption simplifies the computation of the class probabilities, and enables the use of a simple statistical model, which can be easily trained and applied to new images. The core idea of the Naive Bayes Classifiers is to model the probability distribution of the features in an image, given the class label, and to use these distributions to calculate the class probabilities, given the input image. Naive Bayes Classifiers are widely used in computer vision, due to their simplicity, computational efficiency [ZW18], and good performance [ZW18] on many tasks. They are particularly suitable for tasks where the number of features is large and the data is sparse, such as in text classification, or where the dimensionality of the data is high, such as in image classification.

1.2.4.5. Random Forest Algorithms

Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [Bre01]. The essence of the algorithm is that at each iteration, a random selection of variables is made, after which, on this new selection, the construction of a decision tree is started. In this case, "bagging" is performed - a selection of random two-thirds of the observations for training, and the remaining third is used to evaluate the result. This operation is done hundreds or thousands of times. The resulting model will be the result of "voting" the set of trees obtained during the simulation.

1.2.4.6. Linear Regression

Regression analysis is one of basic technique in computer vision. Tradition and ease of computation have made the least squares method the most popular form of regression analysis [MMR91]. The least squares method achieves optimum results when the underlying error distribution is Gaussian. However, the method becomes unreliable if the noise has nonzero-mean components and/or if outliers (samples with values far from the local trend) are present in the data [MMR91]. The outliers may be the result of clutter, large measurement errors, or impulse noise corrupting the data [MMR91]. At a transition between two homogeneous regions of the image, samples belonging to one region may become outliers for fits in the other region [MMR91]. The regression model has three main concepts: efficiency, breakdown point, and time complexity.

Linear regression is a linear model that assumes a linear relationship between input variables (X_i) and a single output variable (Y).

When there is one input variable (x), the method is called simple linear regression. When there are multiple input variables, the statistical literature often refers to the method as multiple linear

regression.

Various methods can be used to prepare or train linear regression. The most common of which is called the Ordinary Least Squares or OLS. The linear equation assigns a scale factor to each input value X. The scale factor is represented by the Greek letter Beta (β). One additional coefficient is also added, adding an additional degree of freedom (for example, moving up and down a two-dimensional area) and is often called the intercept or bias coefficient. The simplest regression problem is when one variable X is input, and there is one output value Y.

1.2.4.7. Other regression algorithms

Besides the logistic regression that was used in object recognition, other experiments were carried out using other algorithms.

ARDRegression

Automatic Relevance Determination (ARD) is based on Bayesian inference method. This is a relatively new method proposed in 1991 by David Mackay [Mac91]. In Automatic Relevance Determination (ARD), we first estimate how much each coefficient (or feature) varies, and then discard those that don't change much by setting them to zero. Think of it like this: we have information about 30 stores (our data points), and for each store, we have 30 different characteristics, like location, size, number of employees, etc. Now, we want to predict something, say, sales volume, using these characteristics. ARD helps us determine which of these characteristics really matter in predicting the sales volume. In this case, it might find that only 5 out of the 30 characteristics are significant, while the other 25 don't contribute much, effectively simplifying our prediction task.

Creating a standard linear regression model in such situations can be challenging. Imagine the actual relationship being described by the formula $Y = w(\times)X + e$, where e is random normal error, and the coefficients w are [1, 2, 3, 4, 5, 0, 0, ...,0]. Here, only the first five coefficients are non-zero, meaning that the features from the 6th to the 30th have no impact on the actual value of Y. But, we're unaware of this. All we have is the data - X and Y - and our task is to determine the w coefficients.

One disadvantage of Automatic Relevance Determination (ARD) is that it can lead to overfitting [GU16]. Additionally, popular update rules used for ARD can be difficult to extend to more general problems of interest or may have non-ideal convergence properties [WN07].

Bayesian Ridge

Bayesian Ridge Regression is a type of regression analysis that can be interpreted as Bayes posterior mean when the prior on the regression parameters is multivariate normal with zero mean and diagonal covariance matrix whose diagonal elements have the same variance/precision [ATT18]. It has been used in various applications such as PV power forecasting [MRA+20].

One of the advantages of Bayesian Ridge Regression is that it introduces additional information to solve ill-posed problems or perform feature selection. This makes it popular in machine learning and statistical modeling more generally [BL20]. The ridge regression parameters have independent and identical Normal priors. The shrinkage parameter, α, is introduced into the model in the form of a hyperparameter [BL20].

DummyRegression

In statistics, dummy regression is a type of regression analysis in which one or more categorical variables are used to predict a continuous dependent variable. Dummy variables are binary variables that take on the values 0 or 1 to represent the presence or absence of a particular category or group.

For example, suppose we have data on the heights and genders of a group of people. We might want to use the gender of each person as a predictor variable to predict their height. To do this, we could create a dummy variable that takes on the value 0 for males and 1 for females. We could then use this dummy variable as a predictor in a regression analysis to see if there is a significant relationship between gender and height.

To perform a dummy regression analysis, we first create a set of dummy variables for each categorical predictor variable in our data. For example, if we have a categorical variable with three categories (A, B, and C), we would create two dummy variables: one for category A and one for category B. The dummy variable for category C would be omitted from the analysis, as it serves as the reference category.

Next, we fit a linear regression model using the dummy variables as predictor variables and the continuous dependent variable as the response variable. We can then interpret the coefficients for the dummy variables as the effect of belonging to a particular category on the dependent variable, holding all other variables constant. For example, if the coefficient for the dummy variable representing category A is significantly different from 0, this suggests that there is a significant relationship between belonging to category A and the dependent variable.

It is important to note that dummy regression should only be used when the categorical variables are not ordinal (i.e., they do not have a natural ordering). If the categorical variables are ordinal, it is better to use ordinal regression techniques.

Lasso

Lasso (Least Absolute Shrinkage and Selection Operator) is a popular method for variable selection and regularization in regression analysis. It can be used to recover an unknown sparse signal from noisy linear measurements by solving an optimization problem that balances the fidelity of the solution with the sparsity of the solution [AAA+22]. This is achieved by minimizing the residual sum of squares subject to the sum of the absolute value of the coefficients being less than a constant. The constant determines the amount of shrinkage applied to the coefficients, with larger values resulting in more shrinkage and therefore more coefficients being set to zero.

LASSO has several advantages over traditional variable selection methods such as stepwise regression. It can handle situations where the number of predictors is larger than the number of observations, and it can also handle correlated predictors. However, LASSO has some limitations as well. For example, it may not perform well when there are strong correlations among predictors or when the true underlying model is not sparse.

There are several variations and extensions of LASSO that have been proposed to address some of its limitations. For example, Elastic Net combines LASSO with ridge regression to handle correlated predictors. Group LASSO and Sparse Group LASSO can be used for structured variable

selection where predictors are grouped into predefined sets.

There are several scientific articles that discuss LASSO and its derivatives in more detail. For example, one article provides a critical review of LASSO and its derivatives for variable selection under dependence among covariates [FFG21]. Another article proposes to employ the Box-LASSO, a variation of the popular LASSO method, as a low-complexity decoder in a massive multiple-input multiple-output (MIMO) wireless communication system[AAA+22]

Lasso-LARS

Lasso-LARS (Least Angle Regression with Lasso penalty) is an algorithm for fitting Lasso models, which are linear regression models with an L1 penalty term to encourage sparse solutions (i.e., solutions with many coefficients set to zero). Lasso-LARS is an efficient algorithm for fitting Lasso models, particularly when the number of predictors is much larger than the number of observations.

Lasso-LARS works by iteratively fitting the Lasso model and adding one variable at a time to the model until all variables are included or the maximum number of variables is reached. At each step, the variable that has the largest absolute correlation with the response variable is added to the model. The coefficient of the added variable is then adjusted so that the model fits the data while still satisfying the Lasso penalty constraint.

The Lasso-LARS algorithm has two main parameters: the Lasso penalty hyperparameter, λ, which determines the strength of the penalty, and the maximum number of variables to include in the model, m. The Lasso-LARS algorithm stops when either all variables have been included in the model or the maximum number of variables has been reached, whichever comes first.

Lasso-LARS has several advantages over other algorithms for fitting Lasso models. One advantage is that it is computationally efficient, particularly when the number of predictors is much larger than the number of observations. Additionally, Lasso-LARS provides a way to easily select the optimal value of the Lasso penalty hyperparameter, λ, by using cross-validation to evaluate the performance of the model at different values of λ.

However, Lasso-LARS has some disadvantages as well. One disadvantage is that it is sensitive to the scale of the predictor variables, and it is generally recommended to standardize the variables before fitting a Lasso-LARS model. Additionally, Lasso-LARS may not perform well when there are many correlated predictors, as it tends to select only one of the correlated variables and set the others to zero. Finally, Lasso-LARS is less efficient at estimating the coefficients of the non-zero variables than other algorithms, such as coordinate descent.

Passive aggressive regression

Passive Aggressive Regression (PAR) is a type of online learning algorithm that is suitable for real-time prediction tasks, including regression. It is a margin-based algorithm that is formulated in terms of deterministic point-estimation problems governed by a set of user-defined hyperparameters [SRO15].

PAR works by iteratively updating the model parameters in response to new data. At each step, the algorithm receives a new data point and makes a prediction based on the current model parameters. The prediction is then compared to the true value, and the model parameters are updated to minimize the prediction error. The update is performed in a passive-aggressive manner: if the

prediction is correct, the model parameters are not changed (passive); if the prediction is incorrect, the model parameters are updated aggressively to correct the error.

One of the advantages of PAR is that it can adapt quickly to changing data distributions. This makes it well-suited for real-time prediction tasks where the data distribution may change over time. However, like all online learning algorithms, PAR may suffer from catastrophic forgetting if the data distribution changes too quickly.

There are several scientific articles that discuss Passive Aggressive Regression in more detail. For example, one article introduces a novel PA learning framework for regression that overcomes some of the limitations of traditional PA algorithms. The authors contribute a Bayesian state-space interpretation of PA regression, along with a novel online variational inference scheme, that not only produces probabilistic predictions but also offers the benefit of automatic hyperparameter tuning [SRO15]. Another article proposes a modified Passive Aggressive Regression model to implement online ensemble forecasting for load forecasting [KWH21].

RANSAC regression

RANSAC (RANdom SAmple Consensus) is an iterative algorithm for fitting a model to data that may contain outliers. It works by selecting a random subset of data, fitting a model to this subset, and then assessing how many other data points fit this model. If enough data points fit, the model is accepted; if not, the process repeats with a new data subset. This approach is particularly advantageous in scenarios with outliers, as RANSAC can identify and exclude them, enhancing the model's accuracy.

The steps to use RANSAC in linear regression are:

1. Select a random subset of data, the "inlier set," with enough points to fit the model.
2. Fit a linear regression model to the inlier set.
3. Determine how many points in the remaining data fit the model, using a maximum allowable error as a threshold.
4. If the number of inliers is better than the current best model, update the model and inlier set.
5. Repeat steps 1-4 until a set maximum number of iterations is reached.

Despite being robust to outliers, RANSAC has its limitations. It can be computationally intensive due to its iterative nature, may not always converge to the true model, especially with a high percentage of outliers, and is sensitive to the choice of the maximum allowable error parameter.

SGD regression

Stochastic Gradient Descent (SGD) is an optimization algorithm that can be used for regression analysis. It works by iteratively updating the model parameters in response to new data. At each step, the algorithm receives a new data point and makes a prediction based on the current model parameters. The prediction is then compared to the true value, and the model parameters are updated to minimize the prediction error.

The update is performed using gradient descent, which means that the model parameters are updated in the direction of the negative gradient of the loss function with respect to the model

parameters. The size of the update is determined by the learning rate, which is a hyperparameter that controls how quickly the model parameters are updated.

SGD has several advantages over traditional batch gradient descent. It can converge faster because it processes one data point at a time instead of computing the gradient over the entire dataset. It can also handle large datasets that do not fit into memory. However, SGD may suffer from slow convergence or getting stuck in local minima if the learning rate is not chosen carefully.

Theil regression

Theil regression is a type of regression analysis that addresses heteroscedasticity, or non-constant variance, which can bias estimates of model parameters. This method models the relationship between a response variable and predictor variables by estimating the local mean of the response variable at each predictor variable value using a weighted average of the response values.

The process of fitting a Theil regression model involves:

1. Specifying the number of nearest neighbors to use in the weighting scheme.
2. Choosing the type of weighting to use, such as uniform or inverse distance weighting.
3. Iteratively estimating the weights and the local mean at each predictor value.

Theil regression is sensitive to the selection of the nearest neighbors and the type of weighting used. It may also be computationally intensive and could struggle in high-dimensional spaces due to the difficulty of accurately estimating the weights.

Tweedie regression

Tweedie regression is a type of generalized linear model that can be used to model data with a Tweedie distribution. The Tweedie distribution is a family of continuous probability distributions that includes the normal, Poisson, gamma, and inverse Gaussian distributions as special cases. It is characterized by its mean, variance, and a power parameter that determines the relationship between the mean and variance.

Tweedie regression models are useful for modeling data with a non-negative response variable that may have a point mass at zero. They can be used to model data with different levels of dispersion, including underdispersion, equidispersion, and overdispersion. Tweedie regression models can be fitted using maximum likelihood estimation or quasi-likelihood methods.

In conclusion, Tweedie regression is a powerful tool for modeling the relationship between a response variable and one or more predictor variables when the response variable follows a Tweedie distribution. It is able to handle a wide range of response distributions and is able to model heteroscedasticity. However, it can be sensitive to the choice of the Tweedie distribution and the power parameter, and may be less interpretable than other generalized linear models.

Perceptron

A Perceptron is a type of artificial neural network that can be used for binary classification. It consists of a single layer of artificial neurons that take a weighted sum of their inputs and apply a non-linear activation function to produce an output. The weights of the inputs are adjusted during training to minimize the prediction error.

Perceptrons were one of the first machine learning algorithms to be developed and have been widely used in pattern recognition and other applications. They are simple to implement and can learn to classify linearly separable data. However, they have some limitations as well. For example, they cannot learn to classify data that is not linearly separable, and they may suffer from slow convergence or getting stuck in local minima.

Perceptrons are simple and easy to implement, but they have some limitations. One limitation is that they can only be used for binary classification. Additionally, perceptrons are not able to model complex relationships between the predictor variables and the response variable. For these reasons, more sophisticated artificial neural networks, such as multi-layer perceptrons and convolutional neural networks, are often used in practice.

1.2.4.8. SIFT Algorithm

SIFT [Low04] was proposed in 2004 by David Lowe, at the University of British Columbia. the scale-invariant feature transform algorithm (SIFT) is used to detect and also describe the local features in a digital image. It locates key points and furnishes them with quantitative information, also known as descriptors used for object detection and recognition.

A descriptor is a key point identifier that distinguishes it from the rest of the mass of singular points. In turn, the descriptors must ensure the invariance of finding a correspondence between singular points with respect to image transformations. In the context of the SIFT algorithm, a key point refers to a point of interest in an image that is invariant to scale and orientation changes. These key points are detected by the algorithm and are used to describe local features in the image. Singular points refer to the same concept as key points. They are distinctive points in an image that can be used for object detection and recognition. The permissible transformations for SIFT include scale changes, rotation, and affine transformations. The algorithm is designed to be invariant to these types of transformations, meaning that it can still detect and describe key points even if the image has been transformed in these ways.

As a result, the following scheme for solving the image matching problem is obtained:

1. Key points and their descriptors are highlighted in the images.

2. According to the coincidence of the descriptors, key points corresponding to each other are selected.

3. Based on a set of matched key points, an image transformation model is built, with the help of which you can get another from one image.

SIFT algorithm has several steps:

1. Scale-space extrema detection: The first stage of computation searches over all scales and image locations. It is implemented efficiently by using a difference-of-Gaussian function to identify potential interest points that are invariant to scale and orientation [Low04].

2. Keypoint localization: At each candidate location, a detailed model is fit to determine location and scale. Keypoints are selected based on measures of their stability [Low04].

3. Orientation assignment: One or more orientations are assigned to each keypoint location based on local image gradient directions [Low04]. All future operations are performed on image

data that has been transformed relative to the assigned orientation, scale, and location for each feature, thereby providing invariance to these transformations [Low04].

4. Keypoint descriptor: The local image gradients are measured at the selected scale in the region around each keypoint [Low04]. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination [Low04].

An important aspect of this approach is that it generates large numbers of features that densely cover the image over the full range of scales and locations [Low04]. A typical image of size 500x500 pixels will give rise to about 2000 stable features (although this number depends on both image content and choices for various parameters) [Low04].

The first stage of keypoint detection is to identify locations and scales that can be repeatably assigned under differing views of the same object [Low04]. Detecting locations that are invariant to scale change of the image can be accomplished by searching for stable features across all possible scales, using a continuous function of scale known as scale-space [AP83; Low04]. The main point in the detection of singular points is the construction of the pyramid of Gaussian and the Differences of Gaussians. A Gaussian (or an image blurred with a Gaussian filter) is an image. Also, an important component is a scalable space. The scalable image space is a set of all possible versions of the original image smoothed by some filter. It is proved [Low04] that the Gaussian scalable space is linear and invariant under shifts, rotations, and the scale does not shift local extrema and has the property of semigroups. In general, scale invariance is achieved by finding key points for the original image taken at different scales. To do this, a Gaussian pyramid is built: the entire scalable space is divided into some sections - octaves, and the part of the scalable space occupied by the next octave is twice as large as the part occupied by the previous one. In addition, when moving from one octave to another, the image is resampled, and its size is halved.

In each image from the DoG pyramid, local extremum points are searched. Each point in the current DoG image is compared to its eight neighbors and to the nine DoG neighbors one level up and down in the pyramid. If this point is greater (less) than all neighbors, then it is taken as a local extremum point.

The next step will be a couple of checks for the suitability of the extremum point for the role of the key one.

First of all, the coordinates of the singular point are determined with subpixel accuracy. This is achieved by approximating the DoG function with a second-order Taylor polynomial [Low04] taken at the calculated extremum point.

After finding the key point, the direction of the point is calculated. The direction of the key point is calculated based on the directions of the gradients of the points adjacent to the key point. All gradient calculations are performed on the image in the Gaussian pyramid, with the scale closest to the scale of the key point.

In the SIFT method, the descriptor is a vector. Like the direction of the key point, the descriptor is calculated on the Gaussian closest in scale to the key point and from the gradients in some key point window. Before calculating the descriptor, this window is rotated by the angle of the direction of the key point, which achieves rotation invariance.

1.2.4.9. Complexity of Algorithms

One of the most important in the study and analysis of the algorithm is to determine the complexity of the algorithm. Big(O) notation is an algorithm complexity metric. It defines the relationship between the number of inputs and the steps taken by the algorithm to process those inputs.

In mathematics, computer science, and related fields, big-O notation is often denoted by a big cryptographic O, which describes the limiting behavior of the function when the argument tends towards a particular value or infinity, usually in terms of simpler functions [DSR11]. BigO notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation [DSR11]. This notation is now frequently used in the analysis of algorithms to describe an algorithm's usage of computational resources: the worst-case or average-case running time or memory usage of an algorithm is often expressed as a function of the length of its input using big O notation [DSR11].

Table 1 shows algorithms in machine learning and their complexity, where m, n, k, v - separate parts of the dataset

Table 1. Complexity of different algorithms

Algorithm	**Complexity**	**Worst complexity**
k-NN	$O(n)$	$O(nm)$
Support Vector Machine	$O(n^2)$	$O(n^3)$
Decision Tree	$O(log(n))$	$O(n)$
k-Means	$O(n)$	$O(n^2)$
Naive Bayes	$O(nd)$	$O(m(n-m+1))$
Random Forest	$O(vnlog(n))$	$O(log(n))$
Linear Regression	$O(k^2(n+k))$	-
CNN	$O(n)$	-
SIFT	$O(mn+k)$	-

1.2.5 Deep learning Computer Vision Algorithms

Computer vision is an interdisciplinary field that has been gaining momentum in recent years (after Convolutional Neural Network), with self-driving cars taking center stage. Another integral part of computer vision is object detection. Object detection helps in pose estimation, vehicle detection, observation, etc. The difference between object detection algorithms and classification algorithms is that in detection algorithms, we try to draw a bounding box around the object of interest to find it in the image. Also, it is not necessary to draw only one bounding box in case of detecting an object, there can be many bounding boxes representing different objects of interest in the image, and we do not know in advance how many.

1.2.5.1. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of Artificial Neural Network (ANN) that is primarily used for image-driven pattern recognition tasks. Image-driven pattern recognition tasks involve the identification of specific patterns or features within images. This can include object

recognition, facial recognition, and scene classification, among others. Pre-processing in CNN requires significantly less compared to other classification algorithms.

CNNs are composed of several different layers (e.g., convolutional layers, downsampling layers, and activation layers) - each layer performs some predetermined function on its input data [WTS+20]. Convolutional layers "extract features" to be used for image classification, with early convolutional layers in the network extracting low-level features (e.g., edges) and later layers extracting more-complex semantic features (e.g., car headlights) [WTS+20].

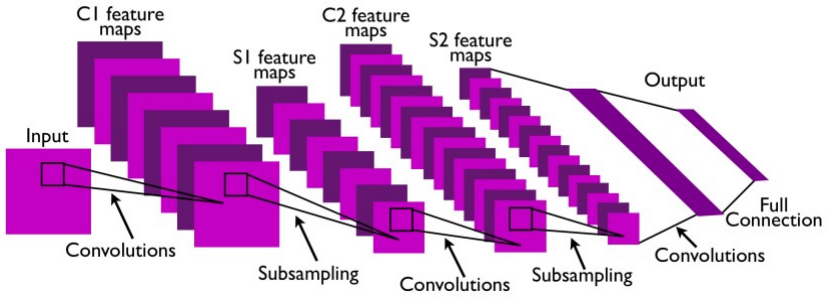


Figure 3. CNN layers [LKF10]

A CNN consists of an input layer, an output layer, and many hidden layers (Fig. 3). These layers perform operations that modify data in order to learn characteristics specific to that data. The three most common levels are convolution, activation or Rectified Linear Unit (ReLU), and pooling.

- Convolution passes the input images through a set of convolution filters, each of which activates certain characteristics of the images.
- The Rectified Linear Unit (ReLU) allows faster and more efficient training by mapping negative values to zero and storing positive values. This is sometimes referred to as activation because only activated characteristics are carried over to the next level. Rectified Linear Unit (ReLU) is defined as *max(0, x)*, which outputs the positive part of its argument. These functions are also often used prior to the output layer to normalize classification scores, for example, the activation function called Softmax performs a normalization on unscaled scalar values, known as logits, to yield output class scores that sum to one [WTS+20].
- Pooling simplifies inference by performing a non-linear reduction in image quality, reducing the number of parameters that networks need to learn.

Pooling has two operations: max-pooling and average-pooling. In most cases, max-pooling is used. The Pooling operation is similar to the convolution operation:

1. The sliding window, usually the (2,2) window, moves across the feature map.
2. From the selected template, the maximum (max-pooling) or average (average-pooling) value is selected.

3. A reduced feature map is generated.

These operations are repeated over dozens or hundreds of layers, with each layer learning to identify different characteristics. Filters are applied to each training image at a different resolution, and the output of each convolved image is used as input to the next layer.

1.2.5.2. R-CNN

The Region-based Convolutional Neural Networks (R-CNN) is a model that facilitates object detection by employing a Convolutional Neural Network (CNN) [GDD+14]. It proposes Regions of Interest (RoI) using a method such as selective search [RvdST+13], and extracts CNN features from these RoIs. R-CNN, Fast R-CNN [Gir15], and Faster R-CNN [RHG+15] use these features to predict object class and bounding box parameters [KST+21].

The R-CNN algorithm operates by detecting potential objects in the image, dividing them into regions, extracting features from each region using CNNs, and then classifying the processed features. While this architecture yields accurate results, it is energy-intensive and requires significant computational power, particularly for large and complex datasets.

Despite being designed for still images, the R-CNN algorithm can be adapted for video data by applying it to individual video frames. However, this process can be computationally intensive and may not fully utilize the temporal information present in the video data.

1.2.5.3. Fast R-CNN

The shortcomings of R-CNN led the authors in 2015 to improve the model. They called it Fast R-CNN [3]. It is based on the following architecture:

The image is fed to the input of a convolutional neural network and processed by selective search. As a result, we have a feature map and regions of potential objects. The coordinates of the regions of potential objects are converted into coordinates on the feature map. The resulting feature map with regions is passed to the RoI (Region of Interest) polling layer. Here, a grid of *HxW* size is superimposed on each region. Then MaxPolling is applied to reduce the dimension. Thus, all regions of potential objects have the same fixed dimension. The resulting features are fed to the input of a fully connected layer, which is passed to two other fully connected layers. The first with the softmax activation function determines the probability of belonging to a class, the second determines the boundaries (offset) of the region of a potential object.

1.2.5.4. YOLO

YOLO is an advanced object detection network developed by Joseph Redmon [RDG+16]. The main thing that distinguishes it from other popular architectures is speed. The YOLO family models are really fast, much faster than R-CNN and others. This means that it is possible to recognize objects in real-time.

In the YOLO architecture, an image is inputted and feature maps are created using its unique CNN, Darknet-53[RF18]. These feature maps are then analyzed by a series of fully connected

layers and a final output layer. This output layer delivers a fixed-size tensor containing information about bounding box positions, sizes, and class probabilities. The tensor is split into a grid, each cell accountable for predicting a fixed number of bounding boxes. For each bounding box, the model predicts the center coordinates relative to the cell location, the box's width and height relative to the image, and an objectness score reflecting confidence in the box containing an object. The model also estimates class probabilities for each bounding box. These predictions are collectively used to generate the model's final output, which is a set of bounding boxes with associated class probabilities.

YOLOv1 divides the input image into an $S \times S$ grid (Fig. 4). If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object [RDG+16]. Therefore, all other cells disregard even the appearance of objects revealed in multiple cells.

Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts [RDG+16]. This confidence score reflects the presence or absence of an object in the bounding box. The confidence score is defined as: $confidence = p(Object) * IOUtruth\ pred$ If no object exists in that cell, the confidence scores should be zero [RDG+16].

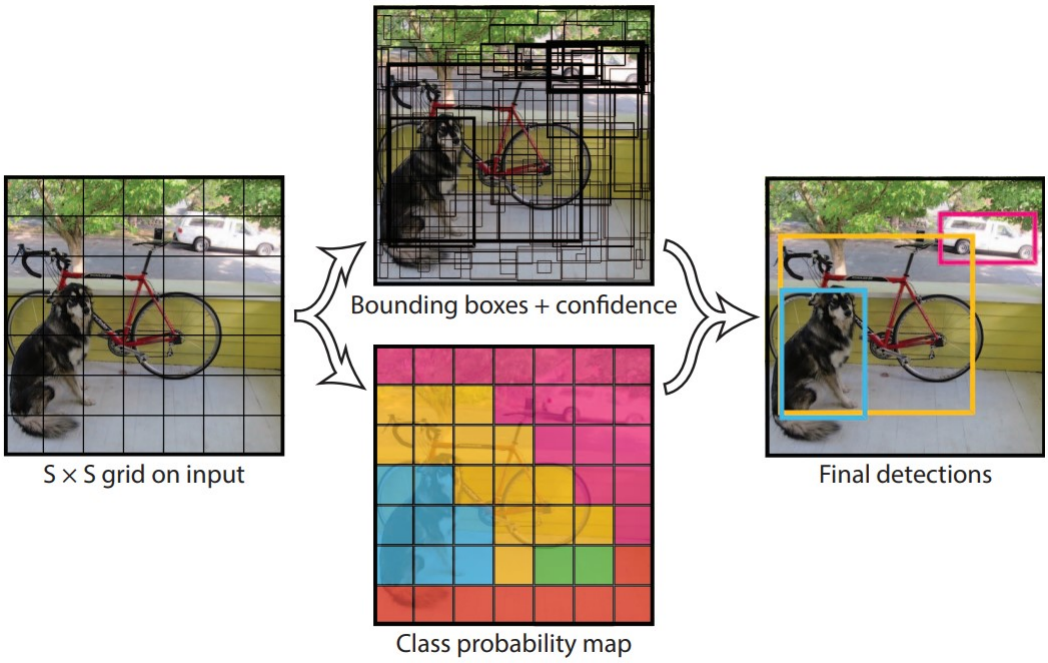


Figure 4. YOLO model 7x7. Image is under such License

Each bounding box consists of 5 predictions: x, y, w, h, and confidence [RDG+16]. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell [RDG+16]. The width and height are predicted relative to the whole image.

1.2.6 Challenges of using Machine Learning in IoT

The key factors considered while implementing machine learning or deep learning with embedded systems are energy consumption, latency, cost, security, privacy, memory space, and processor speed [MK20].

The first big problem is data. The increase in the dimensionality of data in high-dimensional spaces can result in the generation of larger amounts of data, which may pose a challenge for energy efficiency [Hor14]. In the context of machine learning and data analysis, the increase in the dimensionality of data can pose several challenges. One of these challenges is the potential increase in the volume of data that needs to be processed and analyzed. As the number of dimensions (or features) in the data increases, the amount of data required to adequately represent the underlying relationships and patterns in the data can also increase. This phenomenon is known as the "curse of dimensionality" and can result in larger amounts of data being generated and stored.

However, an increase in dimensionality does not always lead to an increase in the volume of data. The relationship between dimensionality and data volume depends on several factors, including the nature of the data, the relationships between the different dimensions, and the methods used to collect and represent the data. In some cases, an increase in dimensionality may actually result in a reduction in data volume if it allows for more efficient representation or compression of the data. Additionally, the programmability of systems used for machine learning can lead to the reading and storing of weights, which can also impact energy consumption. The process of reading and storing weights in machine learning systems can impact energy consumption in several ways. First, the act of reading and writing data to memory requires energy. The amount of energy consumed depends on several factors, including the size of the data being read or written, the speed of the memory, and the efficiency of the memory controller. As the size of the weight data increases, the energy consumption associated with reading and writing the data can also increase.

Second, storing large amounts of weight data can require the use of additional memory resources, which can increase the overall energy consumption of the system. Memory devices consume energy even when they are not actively being accessed, so increasing the amount of memory used by a machine learning system can result in higher energy consumption.

However, it should be noted that the relationship between programmability and energy efficiency is not straightforward, as there are programmable devices that can be designed to optimize energy consumption. The relationship between programmability and energy efficiency in machine learning systems is due to the fact that there are trade-offs involved in designing systems that are both programmable and energy-efficient. Programmability refers to the ability of a system to be easily configured or programmed to perform different tasks. This can be an important feature for machine learning systems, as it allows them to be adapted to different applications and datasets. However, achieving high levels of programmability can sometimes come at the cost of increased energy consumption.

For example, programmable systems may require more complex hardware and software architectures, which can increase the energy consumption of the system. Additionally, the process of reading and storing weights and other data can also consume energy, as discussed in the previous

response.

However, it is possible to design programmable systems that are optimized for energy efficiency. This can involve using specialized hardware and software architectures that minimize energy consumption while still providing a high degree of programmability. The challenge is to find the right balance between programmability and energy efficiency for a given application. Moreover, there exist some problems with poor results, especially if there is a lack of data in the model or a lack of reliable data. A poor result typically refers to a model that has low accuracy or performs poorly on some other evaluation metric. There can be several reasons for poor results in machine learning, including a lack of data or a lack of reliable data. If a model is trained on a small or unrepresentative dataset, it may not be able to accurately capture the underlying relationships and patterns in the data. This can result in poor performance when the model is applied to new data. Similarly, if the data used to train the model is unreliable or contains errors, the model may learn incorrect relationships and produce poor results. If we will consider the first scenario many machine learning algorithms require large amounts of data [MK20] before they start producing useful results. A good example of this is the neural network. Neural networks are data-eating machines that require a lot of training data. The larger the architecture, the more data is required to produce viable results. Reusing data is a bad idea, it's always preferable to have more data. Many machine learning algorithms, including neural networks, require large amounts of data to learn effectively. In some cases, it may be tempting to reuse the same data multiple times to increase the effective size of the training dataset. However, this approach can lead to overfitting, where the model learns to recognize specific characteristics of the training data rather than generalizing to new data. In general, it is preferable to have more data rather than reusing the same data multiple times. This allows the model to learn from a wider range of examples and can improve its ability to generalize to new data.

Another problem can be with privacy and security. Implementing machine learning in the embedded system requires various types and amounts of calculations to be done to analyze the data [MK20]. All these computations are done in the cloud which leads to latency, security, and privacy issues due to sending of data to the cloud as the internet is involved in it [Nad19]. In some situations, an immediate response is needed. An application may not tolerate the time delay occurring due to sending of the cloud [Nad19]. Also, the data may be private and cannot be afforded to be transmitted or shared externally [MK20].

Another challenge is connectivity. IoT-connected devices should have a reliable two-way signaling network as sometimes devices have to collect data from the server or the server has to receive data from devices or sometimes devices have to talk to each other.

2 Methodology

2.1 Settings for conducting experimental studies

This research aims to explore and compare the use of various machine learning and image processing methods for object detection, color detection, MNIST digit recognition, and controlling

LED using the Arduino Nano 33 BLE Sense and other microcontroller platforms such as STM32, Arduino Uno, and Raspberry Pi 4. The experiments are designed to evaluate the performance and feasibility of deploying deep learning models and traditional computer vision methods in low-power and memory-constrained devices, a fundamental aspect of the Internet of Things (IoT) applications.

Three machine learning and image processing methods were primarily used in these experiments: Tensorflow, EloquentTinyML, and OpenCV. The rationale behind choosing these frameworks can be attributed to their ability to run on resource-constrained devices and their wide usage in machine learning and computer vision tasks.

In the first set of experiments, Tensorflow, a powerful deep learning library, was utilized for color and object detection. Here, Tensorflow allowed for the development and deployment of efficient models capable of learning complex patterns from high-dimensional data.

A variation of the first experiment involved recognizing MNIST digits using EloquentTinyML, an open-source library built specifically for running machine learning models on microcontrollers. This exercise allowed for a detailed examination of the performance of specific ML models designed for low-resource environments in recognizing hand-written digits.

The subsequent experiments involved using OpenCV, a widely-used computer vision library, for detecting color and objects. Contrasting to the Tensorflow and EloquentTinyML methods, these experiments did not employ deep learning but relied on classical computer vision methods. This approach offered an opportunity to assess the efficacy and efficiency of traditional image processing techniques in constrained settings.

Finally, the MediaPipe and Arduino experiment explored the possibility of gesture-based LED control. This experiment is of significant importance as it brings human-computer interaction closer to daily life and could have potential applications in smart home systems and interactive installations.

The overarching goal of this research is to provide insight into the capabilities and limitations of current machine learning and image processing methods in constrained environments. By comparing deep learning techniques with traditional methods, the research seeks to guide future development and application of machine learning algorithms in IoT devices. Furthermore, it aims to pave the way for the development of intelligent, low-power, and cost-effective IoT applications that can interact seamlessly with the environment and the users.

2.2 Technological solutions for implementing computer vision implementation in microcontrollers

2.2.1 TinyML

Tiny Machine Learning (or TinyML) is a machine learning technique that combines reduced and optimized machine learning applications that require ”full-stack” solutions (hardware, system, software, and applications), including machine learning architectures, methods, tools, and approaches learning able to run analytics on a device at the very edge of the cloud. TinyML can work with sensors, microcontrollers or low-performance devices. A common IoT approach is to collect data

and send it to a centralized registration server, where machine learning can be used. In order to run a machine learning model on IoT devices, TensorflowLite is used for this. Tensorflow Lite provides small binaries capable of running on low power embedded systems.

2.2.1.1. Tensorflow architecture

In TensorFlow, machine learning algorithms are represented as computational graphs. A computational or dataflow graph is a form of a directed graph where vertices or nodes describe operations, while edges represent data flowing between these operations [Gol16].

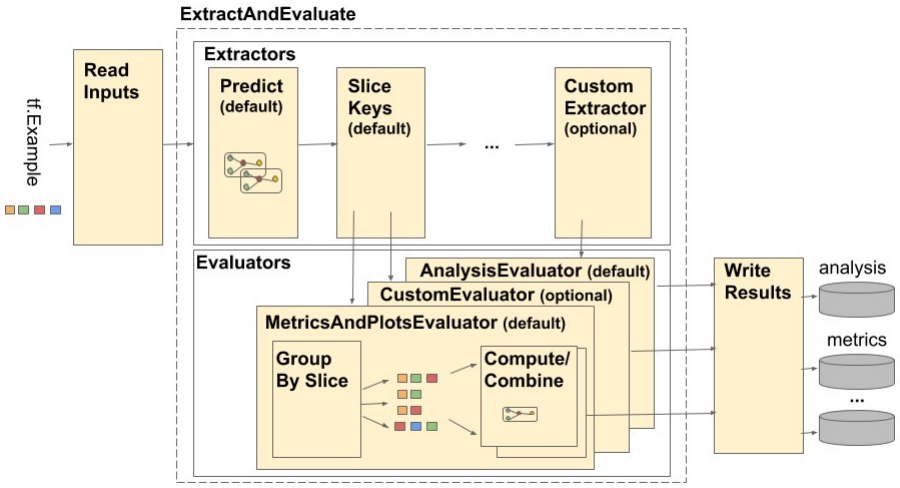


Figure 5. Tensorflow Architecture licensed under(Creative Commons Attribution 4.0 License) [Goo21]

As can be seen in fig. 5 there are four main components:

- Read Inputs takes raw input and converts it into extracts;
- Extractors extract data
- Evaluators take an extract and evaluate it;
- Write Results the evaluation output gets written out to disk. Write Results uses writers to write out the data based on the output keys [Goo16].

A TensorFlow computational graph has different elements such as operations, tensors, variables, and sessions.

1. Operation is a node in a graph that takes zero or more tensor objects as input and produces zero or more tensor objects as output.

2. Tensors are the main object that is manipulated in Tensorflow is the tensor. A tensor is a generalization of vectors and matrices to higher dimensions. Inside TensorFlow, tensors are represented as n-dimensional arrays of basic data types. Tensor has the following options: data type (*float32*, *int32*, or *string*, for example) and dimensions. All elements of a tensor have the same

data type, and it is always known. The dimensions (the number of dimensions and the size of each dimension) can only be partially known.

3. Variables are the best way to represent the public, stored state. In the context of TensorFlow, a *tf.Variable* is a specific type of tensor whose value can be changed by performing operations on it. This is different from the more general concept of a variable in computer science, which refers to a named storage location that can hold a value. A *tf.Variable* is used to represent shared, persistent state that can be manipulated by the TensorFlow computational graph. It stores a constant tensor internally, and certain operations allow the values of this tensor to be read and modified. These changes are visible across multiple *tf.Sessions*, meaning that multiple threads or processes can see the same values for a *tf.Variable*. Variables are used through the *tf.Variable* class. *tf.Variable* is a tensor whose value can be changed by performing operations on it. Internally, *tf.Variable* stores a constant tensor. Certain operations allow us to read and change the values of this tensor. These changes are visible across multiple t*f.Sessions*, so multiple executing threads can see the same values for a *tf.Variable*.

4. Sessions are in TensorFlow, the execution of operations and evaluation of tensors may only be performed in a special environment referred to as a session. One of the responsibilities of a session is to encapsulate the allocation and management of resources such as variable buffers [Gol16].

2.2.2 EdgeImpulse

EdgeImpulse is a solution that aims to simplify the deployment of machine learning applications on embedded devices based on the Cortex-M core by collecting real-world sensor data, training ML models with that data in the cloud, and then deploying the model to an embedded device. Edge Impulse uses a variety of algorithms to optimize machine learning models for deployment on embedded devices. For example, Edge Impulse uses UMAP, a dimensionality reduction algorithm, to project high dimensionality input data into a 3-dimensional space [Edg22]. Additionally, Edge Impulse uses TensorFlow's Model Optimization Toolkit to quantize models, reducing their weights' precision from *float32* to *int8* with minimal impact on accuracy [Sit21].

EdgeImpulse is a cloud-based solution that has a SaaS (Software as a Service) type of model.

EdgeImpulse uses several rules to create a model:

1) when collecting data, the data must be sent to the EdgeImpulse cloud service using Data forwarder. The data forwarder is used to easily relay data from any device to Edge Impulse over serial. Devices write sensor values over a serial connection, and the data forwarder collects the data, signs the data, and sends the data to the ingestion service. The data forwarder is useful to quickly enable data collection from a wide variety of development boards without having to port the full remote management protocol and serial protocol, but only supports collecting data at relatively low frequencies. The remote management server implements a two-way protocol between devices and Edge Impulse, which allows users to control devices (for example, to acquire new data) straight from the studio. Devices can either connect directly to the remote management service over a websocket (see the protocol on this page), or can connect through a proxy. The ingestion service is used to send new device data to Edge Impulse. Data Acquisition format is a small specification that describes

the type of data, the sensor data itself, and information about the device that generated the data.

2) After training the model, the model can be sent back to the device. It can be sent in three different ways.

2.1) the model can be separately exported to the project framework.

2.2) using the company's cloud services can be downloaded to the device itself.

2.3) using webassembly can be uploaded to a server or computer.

2.2.3 Other existent libraries for microcontrollers

MicroML is a project to bring machine learning algorithms to microcontrollers. It was born as an alternative to TensorFlow for microcontrollers, which is solely dedicated to artificial neural networks, MicroML can be run even on 8-bit microcontrollers. The library is written only in Python and uses different machine learning algorithms such as Support Vector Machine, linear regression, etc. This library has a built-in parser that converts the entire process into a low-level C language and creates a model file that can later be connected to the microcontroller. The advantage of the library is that it can be used together with libraries that are designed for statistics and probability, such as scikit-learn [PVG+11] or NumPy [HMvdW+20].

The disadvantage of this library may be that tests have not been carried out by the developer on different models of a microcontroller, for example, Arduino Nano, it is also not clear how many resources the model consumes after conversion.

There are many different libraries such as *MicroML*, *emlearn*, *EloquentTinyML*. All these libraries are united by one principle, models are trained on machines with high computing power, and the model is converted into a low-level language and used later on the microcontroller.

2.2.4 OpenCV

OpenCV, which stands for Open-Source Computer Vision Library, is a widely-used and well-documented open-source library that incorporates numerous computer vision algorithms. It was developed to provide a common infrastructure for computer vision applications, with the goal of accelerating the use of machine perception in commercial products. OpenCV is highly efficient computationally, which assists with real-time applications. The library provides a wide range of algorithms for various computer vision tasks, including many machine learning algorithms like Bayes Classifier, K-Nearest Neighbors, Support Vector Machines, Decision Trees, and more. These are housed in a module called 'ml'.

Images processed through OpenCV are primarily composed of picture elements known as pixels [CAP+12]. These images can easily contain tens of thousands of pixels or even more since modern images have a high resolution, which makes efficient pixel processing crucial [CAP+12]. OpenCV handles this by treating images as matrices, where each element of the matrix corresponds to one pixel [CAP+12]. This efficient data structure allows OpenCV to effectively manage and process the high-volume data associated with computer vision tasks.

An image is like a grid where each square, or pixel, holds a specific value. For instance, in a 400x300 pixel image, we have 120,000 individual pixels. Pixels can be either grayscale or RGB.

In grayscale, pixel values range from 0 (black) to 255 (white), with in-between values representing various shades of gray.

OpenCV allows to create a matrix of various value types (as described earlier) and it is important to note that certain operations (pixel processing) can be applied only to certain matrix types [CAP+12]. Moreover, certain image processing could be more effective if appropriate color space is chosen [Dub10].

2.2.5 Tensorflow

TensorFlow is an open-source software library for dataflow and differentiable programming across a range of tasks. It is commonly used for machine learning applications such as neural networks. One type of neural network that can be built using TensorFlow is a Convolutional Neural Network (CNN), which is often used for image recognition tasks. CNNs are designed to take in input data in the form of images and process the data through multiple layers, each of which applies a different set of filters to the image to extract different features. These features are then used to automatically recognize and classify the images. TensorFlow also helps you to build a neural network model [VTP+20], which is capable of automatically recognizing images. These are typically Convolutional Neural Networks (CNN). There are two types to implement TensorFlow image recognition:

Classification: Train the CNN to recognize classifications like cups, books, blades, vehicles, or whatever else. The framework arranges the picture in general, depending on accessible classes [KSH12; LWX+15].

Object Detection: This procedure is fit for distinguishing numerous items from an equivalent picture at the same time. It ought to likewise label the object and finds its area inside the picture and its exactness individually.

There are three models in TensorFlow, which respectively are the calculation model: calculation graph, the data model (tensor), and the running model (session). The data in TensorFlow is represented by a Tensor data structure, Flow represents the flow and calculation of data [YLJ18]. It can use the feed (or fetch) to assign (or get) the data in the tensor [YLJ18].

TensorBoard is a visual tool corresponding to the TensorFlow calculation graph. It can visualize the output log files during the running process of the TensorFlow program, and effectively display the calculation graph and the trend of various parameter indicators with time during operation in TensorFlow, and facilitate the understanding and debugging of the program [YLJ18].

2.2.6 Microcontrollers

A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. It's essentially a small, low-cost computer on a single chip that includes a processor core, memory (RAM), and programmable input/output peripherals. Commercial and developer boards are the two types of microcontroller boards, with commercial boards being designed for specific tasks, while developer boards are intended for general-purpose use. To maximize flexibility for testing and developing various applications, we will use a developer board for this project, thereby eliminating the need for designing a custom PCB.

To successfully develop a schematic diagram, it is needed to immediately make a list of materials that will be used in its construction. In the experimental study of this work, 4 microcontrollers are used, and table 2 shows the technical characteristics of the microcontrollers used.

As can be seen from table 2, all 4 microcontrollers have different memory sizes, different processors, as well as different voltage consumption. According to these characteristics, these devices were selected.

All these boards possess adequate computational power and memory capacity, enabling their use in fundamental TinyML applications, and are further backed by the TensorFlow Lite framework. TinyML refers to the deployment of machine learning models on low-power microcontrollers. These boards have enough resources to run basic machine learning models and are supported by the TensorFlow Lite framework, which is designed to run machine learning models on resource-constrained devices.

Table 2. Characteristics of microcontrollers

Characteristics	**Arduino Uno**	**Arduino Nano 33 BLE Sense**	**STM32L053R8**	**Raspberry Pi** 4
Processor	Xtensa LX6	ARM Cortex-M4	ARM Cortex-M0+	ARM Cortex-A72
Clock Speed	160MHz	64MHz	32MHz	1.5GHz
Flash Memory	32MB	1MB	64kB	32GB
SRAM	520kB	256kB	8kB	2GB
Voltage	5V	3.3V	3.6V	5V
Digital I/O	9	14	51	40
Analog Pins		8		

2.2.7 Choosing a software development environment

After creating the basic schemes of the device and its construction in the material version, you can proceed to the development of software for the device.

Visual Studio Code was chosen as the development environment, and Platformio was used as a framework instead of using Arduino IDE or the development environment for STM32. Comparing Visual Studio Code with Arduino IDE or STM32 development environment, VSCode is a lightweight IDE and has an extension system that provides rich features to the code editor. This allows developers to customize their development environment with extensions for different programming languages and frameworks. Additionally, VSCode has a file manager tool that makes it easier to work with multi-file projects as well as integrated Git Version Control and integrated Github Syncing. Another useful feature is Code IntelliSense which can scan the project dependencies, locate where they are and find code references from all those files.

Among the advantages of Platformio:

- Intuitive choice of microcontroller to quickly identify the parts that meet requirements

- Advanced debugging options such as:
  - Local, Global, and Static Variable Explorer;
  - Conditional Breakpoints; Expressions and Watchpoints;
  - Generic Registers;
  - Peripheral Registers;
  - Memory Viewer;
  - Disassembly;
  - Multi-thread support;
  - A hot restart of an active debugging session
- Ability to create unit tests to check the correctness of the program
- Remote development
- Library Manager for the hundreds popular libraries

Table 3 provides a comparison of Platformio, Arduino IDE and STM32CubeProgrammer.

Table 3. Comparison between IDEs

Platformio	**Arduino IDE**	**STM32CubeProgrammer**
Has device management system for different platforms such as Arduino Uno, ESP8266, STM32, etc.	Has device management system only for Arduino devices such as Arduino Uno, Arduino Mega, Arduino Leonardo, etc.	Has device management system only for STM32 types of device.
Has own dependency management system	Has own dependency management system	Does not have
Provides help such as auto-complete	Does not provide auto-complete	Provides auto-complete
Can be integrated with code repository such as Github [CT22]	Cannot be integrated with code repository such as Github [CT22]	Cannot be integrated with code repository such as Github [CT22]

3 Results of experimental investigation

3.1 Use of TensorFlow library

3.1.1 Detecting color using deep learning methods

The inspiration and foundational groundwork for this experiment originated from a blog post on the Arduino official site, which can be found at blog ”Fruit identification using Arduino and TensorFlow”. This blog discussed the concept of fruit identification using Arduino and TensorFlow, which introduced the idea of integrating Deep Learning algorithms for robust color detection.

I expanded on this concept by developing a comparative method using OpenCV. This classic method was developed by me as an alternative and comparison to the Deep Learning approach. The method focuses on image processing and feature extraction techniques. While the Deep Learning approach can handle varying conditions and learn from vast datasets, OpenCV-based approach provides a more lightweight, rapid, and less resource-intensive solution. Deep learning algorithms can handle the complexities and variations in color appearance and provide accurate color detection results. The Arduino Nano 33 BLE Sense board has the necessary hardware to run such deep Learning algorithms, making it a suitable platform for developing color detection applications.

For this project was considered Arduino Nano 33 BLE Sense board to measure the main component of the light projected using the on-board APDS9960 sensor [Tec15]. The APDS9960 sensor is a multipurpose device that features advanced Gesture detection, Proximity detection, Digital Ambient Light Sense (ALS) and Color Sense (RGBC).

White light is made of all of the colors of the rainbow because it contains all wavelengths, and it is described as polychromatic light. The color light is separated in four different channels (Red, Green, Blue, and Clear light intensity). The Clear light intensity channel measures the overall intensity of light, regardless of its color. This channel is not directly related to color, but it can provide useful information about the lighting conditions in a scene. For example, if the Clear light intensity is low, it may indicate that the scene is poorly lit, which can affect the accuracy of color detection. By combining the information from the Clear light intensity channel with the information from the Red, Green, and Blue channels, it is possible to more accurately determine the colors in a scene and compensate for variations in lighting conditions. Each of them has a UV (ultraviolet) and IR (infrared) blocking filter and a dedicated data converter to read the data simultaneously. UV and IR blocking filters are used to prevent UV and IR light from reaching the sensor. This is important because UV and IR light can interfere with the accuracy of color detection. The human eye is not sensitive to UV or IR light, so these wavelengths are not visible to us and do not contribute to our perception of color. However, many digital sensors are sensitive to UV and IR light, and if these wavelengths are not blocked, they can affect the sensor's readings and cause the colors in an image to appear distorted. By using UV and IR blocking filters, it is possible to ensure that only the visible wavelengths of light reach the sensor, improving the accuracy of color detection.

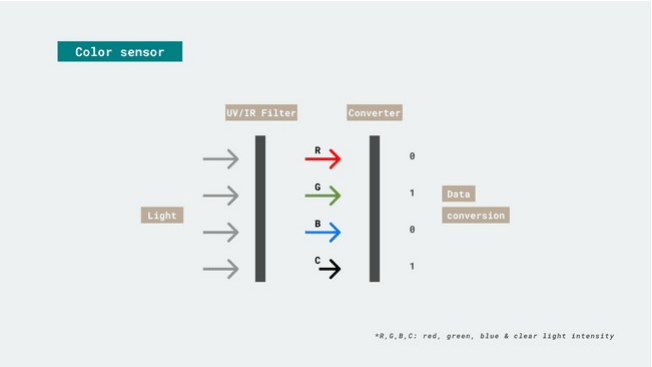


Figure 6. The principle of operation of the color sensor [Tro19]

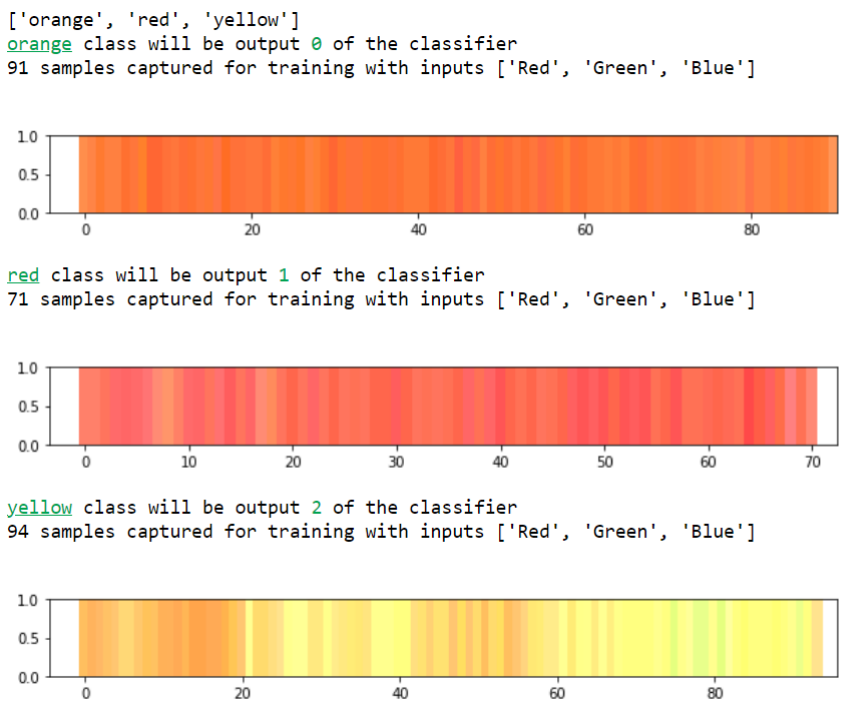


Figure 7. Color range of colors

CSV files were employed for color recognition, utilizing the APDS9960 sensor to capture color data. The APDS9960 is an advanced digital RGB, ambient light, and proximity sensor that integrates a wide range of features within a single compact package. It enables precise and reliable color detection by leveraging its RGB color sensing capability.

The given data table adheres to the CSV (Comma-Separated Values) format, signifying a dataset comprising three distinct columns: Red, Green, and Blue. Each row within the table corresponds to a collection of values representing these respective color channels.

As part of the color classification process, a rapid sampling approach was implemented, utilizing only a single example per class. This method allows for a swift proof of concept, enabling initial detection and classification of objects based on their color attributes.

Here's a description of the table:

- Column 1: Red This column represents the intensity or value of the red color channel for each entry in the dataset.
- Column 2: Green This column represents the intensity or value of the green color channel for each entry in the dataset.
- Column 3: Blue This column represents the intensity or value of the blue color channel for each entry in the dataset.

The next is to parse csv file and transforms them to a format that will be used to train the full connected neural network. The full connected neural network mentioned in the before refers to a neural network architecture that consists of fully connected layers, also known as dense layers. In the code, after parsing and preparing the dataset, the inputs and outputs are processed to be used for

training the neural network. The inputs are stored in the inputs list, and the corresponding outputs are stored in the outputs list. The dataset is then randomized using *np.random.shuffle* to ensure an even distribution of data for training, testing, and validation. The randomized inputs and outputs are split into three sets: training, testing, and validation, using the *np.split* function. The resulting *sets*, *inputs_train*, *inputs_test*, *inputs_validate*, *outputs_train*, *outputs_test*, and *outputs_validate*, are used to train, test, and validate the full connected neural network. For this, the pandas library was used, which has the *read_csv* function. After constructing and training the TensorFlow model using the high-level Keras API [Ker20] on a computer, the model was subsequently uploaded to the Arduino for further calculations. The Keras API provides a convenient and efficient way to develop and refine machine learning models. With its built-in functions and simplified syntax for creating and configuring layers, users can rapidly build and refine models with ease and efficiency. The *tf.keras* module offers first-class support for TensorFlow-specific functionality, including fast execution, *tf.data* collections, and Estimators. In Keras, layers are collected to build models, usually in the form of a layer stack called a *tf.keras.Sequential* model.

Below is an example for building model:

```
....
# Split the recordings (group of samples) into three sets:
    training, testing and validation
TRAIN_SPLIT = int(0.6 * num_inputs)
TEST_SPLIT = int(0.2 * num_inputs + TRAIN_SPLIT)

inputs_train, inputs_test, inputs_validate = np.split(inputs, [
    TRAIN_SPLIT, TEST_SPLIT])
outputs_train, outputs_test, outputs_validate = np.split(outputs,
     [TRAIN_SPLIT, TEST_SPLIT])
....
# build the model and train it
model = tf.keras.Sequential()
# relu is used for performance
model.add(tf.keras.layers.Dense(8, activation='relu'))
model.add(tf.keras.layers.Dense(5, activation='relu'))
# softmax is used, because we only expect one class to occur per
    input
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'
    ))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
history = model.fit(inputs_train, outputs_train, epochs=400,
    batch_size=4, validation_data=(inputs_validate,
    outputs_validate))
```

A Sequential model is appropriate for a plain stack of layers where each layer has exactly one

input tensor and one output tensor. Data flows sequentially through each layer until it reaches the final output layer. The first layer to place into a neural network is the input layer. Layers after the first one do not need to specify the input dimension, since after the input layer it will be the representation of the input in terms of weights and biases that will pass from one layer to another.

The next step is to convert the model to TFlite format and put it into Arduino and calculate mean squared error. Fig. 8 shows the result of the detection three colors.

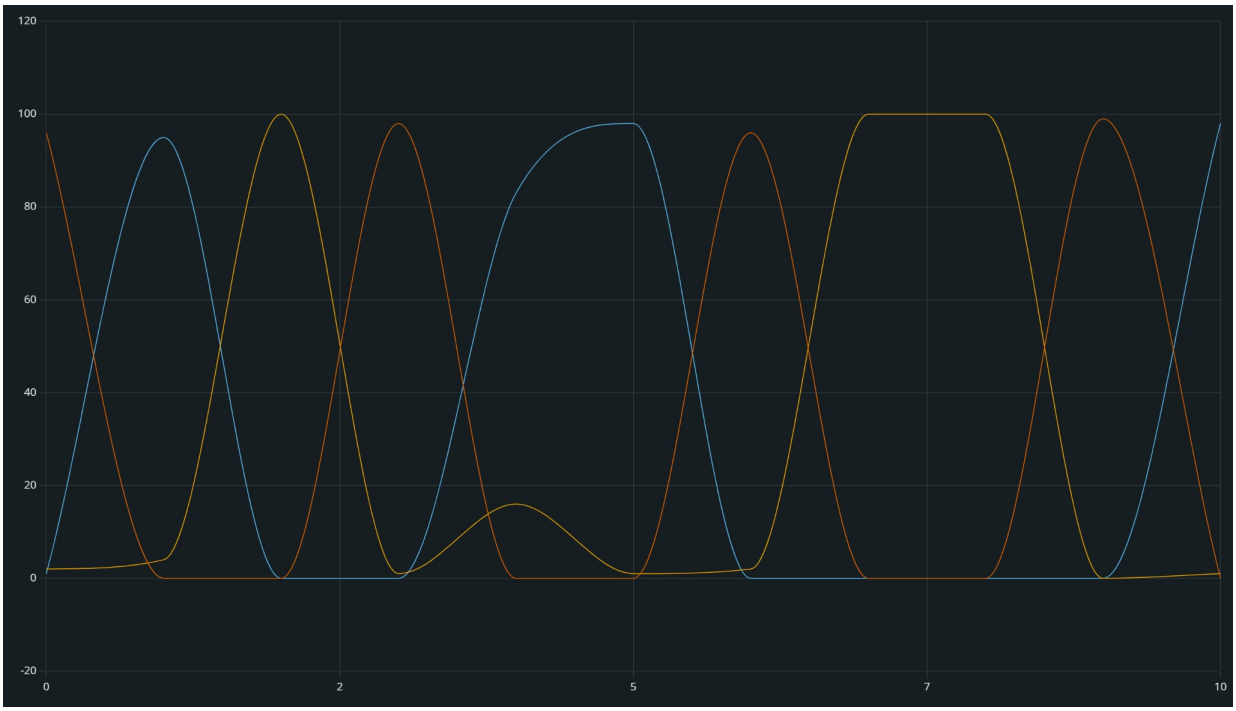


Figure 8. Detection three colors, where yellow line presents yellow color, orange - orange and red - blue. The range between 0 to 100 - a percentage of detection. The vertical Y axis auto adjusts as the value of the output increases or decreases, and the X axis is a fixed 500-point axis with each tick of the axis equal to an executed serial println command

In the referenced illustration (Fig. 9), the project utilized an optimization algorithm known as RMSprop.

Classification Report

	precision	recall	f1-score	support
orange	1.00	1.00	1.00	11
red	1.00	1.00	1.00	17
yellow	1.00	1.00	1.00	23
accuracy			1.00	51
macro avg	1.00	1.00	1.00	51
weighted avg	1.00	1.00	1.00	51

Figure 9. The results of accuracy, precision, recall and F1 using RMSprop optimizer

The Root Mean Square Propagation (RMSprop) optimizer is a sophisticated algorithm for gradient descent, particularly useful in dealing with non-convex optimization problems. It employs a moving average of squared gradients to normalize the gradient itself. Essentially, RMSprop modulates the learning rate for each weight in the model based on the recent magnitudes of its gradients.

This property makes RMSprop particularly suitable for dealing with both plateaus and noisy updates.

The exploration of the project's results will now extend to include different types of optimizers, namely Stochastic Gradient Descent (SGD) Fig. 10, Nadam (Fig. 11), and Adam (Fig. 12).

Stochastic Gradient Descent, or SGD, is an optimization technique that seeks to minimize a given function by iteratively moving in the direction of steepest descent, defined by the negative gradient of the function at the current point. Unlike standard gradient descent, SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

Classification Report

	precision	recall	f1-score	support
orange	0.88	0.94	0.91	16
red	0.90	0.82	0.86	11
yellow	1.00	1.00	1.00	24
accuracy			0.94	51
macro avg	0.93	0.92	0.92	51
weighted avg	0.94	0.94	0.94	51

Figure 10. The results of accuracy, precision, recall and F1 using SGD optimizer

Nadam (Nesterov-accelerated Adaptive Moment Estimation) is an optimizer that combines the Adam and Nesterov momentum techniques. It leverages the benefits of Adam's adaptive gradient methods, which perform well in scenarios with large-scale data or parameters, and incorporates the anticipatory characteristics of Nesterov momentum to increase responsiveness to changes.

Classification Report

	precision	recall	f1-score	support
orange	1.00	1.00	1.00	16
red	1.00	1.00	1.00	11
yellow	1.00	1.00	1.00	24
accuracy			1.00	51
macro avg	1.00	1.00	1.00	51
weighted avg	1.00	1.00	1.00	51

Figure 11. The results of accuracy, precision, recall and F1 using Nadam optimizer

Adam (Adaptive Moment Estimation) is another popular optimizer that calculates adaptive learning rates for different parameters. It combines the perks of two extensions of SGD—RMSprop and AdaGrad—by computing adaptive learning rates for each weight while maintaining an exponentially decaying average of past gradients, similar to momentum.

Classification Report

	precision	recall	f1-score	support
orange	0.93	0.88	0.90	16
red	0.77	0.91	0.83	11
yellow	1.00	0.96	0.98	24
accuracy			0.92	51
macro avg	0.90	0.91	0.91	51
weighted avg	0.93	0.92	0.92	51

Figure 12. The results of accuracy, precision, recall and F1 using Adam optimizer

The results from different optimization algorithms can vary due to the unique ways each algorithm navigates the optimization landscape to find the minimum of the loss function.

RMSprop and Nadam optimizers resulted in perfect classification, as evidenced by precision, recall, and f1-score values of 1.00 for each category. This suggests that these optimizers could efficiently find the optimal set of weights for the model to achieve maximum predictive performance.

In contrast, the SGD and Adam optimizers showed slightly lower performance. The nature of SGD, which utilizes a single randomly selected sample at each step, can lead to more noise during optimization and a longer convergence time. This might explain why it could not reach the same perfect classification as RMSprop or Nadam. Adam optimizer, which also has an adaptive learning rate like RMSprop, showed good results, but not perfect. This might be due to the specific configuration of hyperparameters, such as the learning rate, or the characteristics of the dataset itself.

Nadam optimizer combines the benefits of both Adam and Nesterov momentum, and in this case, it seems to perform as well as RMSprop, showing perfect classification.

The optimal choice of an optimizer may depend on the specific problem at hand, the dataset characteristics, and the computational resources available. While RMSprop and Nadam have shown superior performance in this instance, it might not always be the case with different datasets or model architectures.

However, based on this data and these results, RMSprop and Nadam appear to be the superior optimizers for this specific task as they have achieved perfect classification accuracy, precision, recall, and F1 scores.

```
loss: 0.2219 - mae: 0.4438 - val_loss: 0.2212 - val_mae: 0.4430
```

Figure 13. The result of mean absolute error and loss for RMSprop

The mean squared error is 0.0062 with 400 epochs.

The color detection process involves the identification of specific colors in an image. The traditional approach of decomposing the image into its Red, Green, and Blue (RGB) spectral components may provide a simple representation of the image's color information, but it fails to account for factors such as ambient lighting, camera parameters, and object surface reflectance. This is because the RGB color model is a device-dependent color model, meaning that the colors represented by the RGB values depend on the device used to capture or display the image. Factors such as ambient lighting and camera parameters can affect how colors are captured by a camera and represented in an image. Similarly, the surface reflectance of an object can affect how its colors appear in an image. These factors can cause variations in the appearance of colors in an image that are not accounted for by the simple RGB representation. These factors can significantly alter the RGB values of an object and make it difficult to perform accurate color detection using simple algorithmic decomposition.

Computer Vision (CV) and Machine Learning (ML) approaches, on the other hand, can handle these complexities and provide more robust color detection results. For example, color constancy algorithms can be used to estimate the color of the light source and correct the colors in an image to appear as they would under a standard light source. This can help to reduce the effects of am-

bient lighting on the appearance of colors in an image. Additionally, machine learning algorithms can be trained on large datasets of images taken under different lighting conditions to learn how to recognize and classify objects based on their colors, even when the lighting conditions vary. These approaches can provide more robust color detection results by accounting for the effects of ambient lighting on the appearance of colors in an image. CV and ML algorithms can analyze multiple features of an image and apply advanced mathematical models to accurately identify the desired color, even in challenging conditions. These features can include color values, texture, shape, and spatial relationships between objects in the image. The number of features analyzed by a CV or ML algorithm can vary depending on the specific algorithm and the complexity of the image being analyzed. The accuracy of a CV or ML algorithm can be measured using metrics such as precision, recall, and F1 score. These metrics can be used to compare the performance of different algorithms or to evaluate the performance of a single algorithm on different datasets. The term ”advanced mathematical models” in the context of Computer Vision (CV) and Machine Learning (ML) algorithms refers to a collection of computational and statistical methods used for tasks such as pattern recognition, classification, and prediction. Here's how they apply to image analysis:

1. Feature Extraction: CV and ML algorithms often start by extracting features from an image. This can involve identifying edges, textures, shapes, and colors within an image. Techniques such as convolutional layers in Convolutional Neural Networks (CNNs) or transformation methods like Fourier and Wavelet Transformations can be employed for this.

2. Dimensionality Reduction: High-dimensional data can be difficult to process and visualize. Algorithms such as Principal Component Analysis (PCA) or t-distributed Stochastic Neighbor Embedding (t-SNE) are used to reduce the dimensionality of the data while preserving its structure. This simplifies the computational requirements of subsequent steps and can also improve the performance of the algorithms.

3. Classification and Regression: After feature extraction and potential dimensionality reduction, the processed data is used for tasks such as classification or regression. For instance, Support Vector Machines (SVMs), Decision Trees, Random Forests, and Neural Networks are commonly used for classification tasks, while regression tasks might use Linear Regression, Logistic Regression, or variants of Neural Networks.

These algorithms can also be trained to improve their color detection accuracy over time, making them ideal for applications that require high-precision color detection. Machine Learning (ML) algorithms can be trained to improve their color detection accuracy over time. This involves adjusting the parameters of the mathematical model used by the algorithm to better fit the data. As the algorithm is exposed to more data, it can learn to make more accurate predictions about the colors present in an image.

The training process typically involves dividing a dataset into a training set and a validation set. The algorithm is trained on the training set and its performance is evaluated on the validation set. The parameters of the mathematical model are adjusted to minimize the error between the

predicted and actual colors in the validation set. This process is repeated until the performance of the algorithm on the validation set reaches an acceptable level.

3.1.2 Object detecting using classical method

The original inspiration for this experiment and idea was taken from the following blog post: ”Detecting Pokemon with Arduino and TinyML”. This method employed a logistic regression model trained in TensorFlow to detect two specific Pokemon - Pikachu and Bulbasaur. While Logistic Regression can be quite effective in many cases, it’s just one of many existent regression models. For a more comprehensive evaluation and to potentially increase the accuracy of predictions, it can be beneficial to explore other regression models, such as Linear Regression, Polynomial Regression, etc.

In practice, a user can present a Pokemon in front of the color sensor, which will subsequently generate RGB values. These values are then utilized by the model to make a prediction as to whether the presented object is Pikachu or Bulbasaur. If the prediction concludes Pikachu, the Arduino LED shines yellow; if it is Bulbasaur, it turns green. The predicted label is also printed for confirmation.

For data collection purposes, the program reads the RGB intensity values from the sensor and prints them as a seven-element tuple. The first three elements of this tuple represent the RGB values; the subsequent three elements showcase the RGB ratios, derived by dividing the color values by the sum of all the values. The final element is the class label, which can either be ”Pikachu” or ”Bulbasaur”.

However, it should be noted that the RGB intensity values collected from the sensor are not a true reflection of the scanned object’s actual RGB colors. To scale the values appropriately to the RGB scale, the program utilizes the MinMaxScale function from the Scikit-Learn library.

The *draw_colors()* function is also applied to generate a rectangle (using *cv2.Rectangle()*), where each entry represents a color. The loops responsible for the creation of this rectangle iterate over the sorted and reversed DataFrame to draw the colors from the darkest (255) to the lightest (0), as depicted in Fig. 14.

Contrastingly, I decided to conduct a comparative study using a classical method that I developed using OpenCV. In the forthcoming sections, I will discuss this method in detail.

Figure 14. Color histogram

These visualizations show their range and relationship. Pikachu’s primary color is yellow, a color obtained from red and green Fig. 14. That’s why the graph shows a linear relationship between these tones Fig. 15.

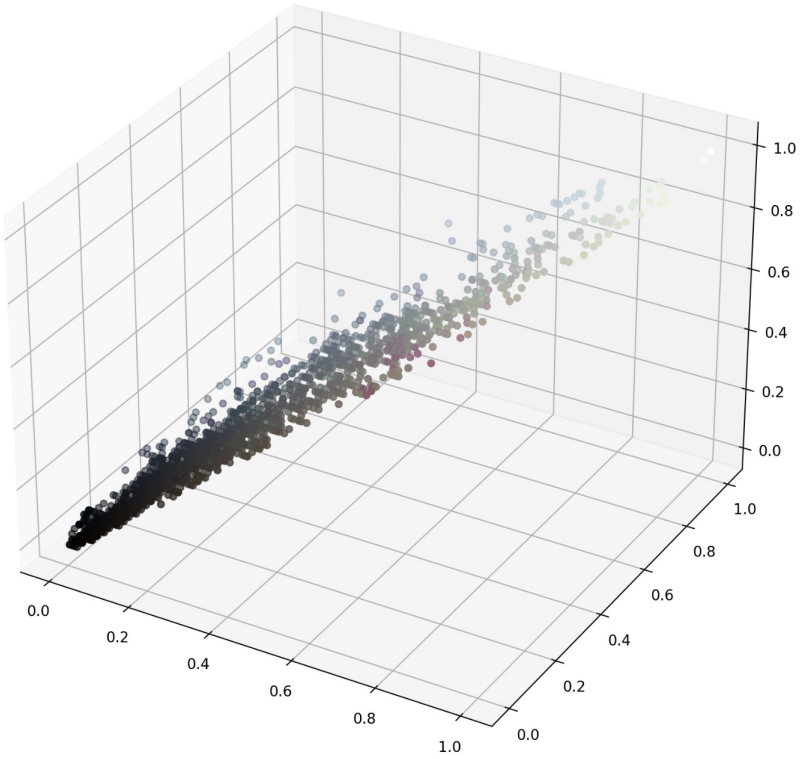


Figure 15. 3d cube upscaled color

Classifier is a logistic regression model trained with TensorFlow in Python. The network has a layer of one unit, an input shape of 3 (because we have three features), and it's trained with an Adam optimizer [KB15] with a learning rate of 0.01 and a binary cross-entropy loss function. Since the activation is sigmoid, its output is a number between 0 and 1 where 0 is Pikachu, and 1 is Bulbasaur; in a logistic regression model, we usually set the threshold at 0.50, meaning that any value under it is the 0 class and those equal or greater than, is the 1 class.

```
....
pikachu_vals_scaled = pd.DataFrame(pikachu_scaler.fit_transform(
    pikachu_vals), columns=pikachu_vals.columns)
bulbasaur_vals_scaled = pd.DataFrame(bulbasaur_scaler.
    fit_transform(bulbasaur_vals), columns=bulbasaur_vals.columns)
....
## Train
X = pd.concat([pikachu[['RedRatio', 'GreenRatio', 'BlueRatio', '
    Class']], bulbasaur[['RedRatio', 'GreenRatio', 'BlueRatio', '
    Class']]])
y = X.Class
X = X.drop('Class', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=0)
```

```
clf = LogisticRegression(random_state=0).fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
preds = [1 if x > 0.50 else 0 for x in y_pred]
print(classification_report(y_test, preds))
```

After model is trained we must put it into the Arduino. Model cannot be used on the Arduino as it is. It must be converted to a TensorFlow Lite model and then encode it as a byte array in C++.

3.1.2.1. Data description

To ensure the foundational requirements of machine learning, data collection was initiated as the initial step. A Sketch was developed to procure the training data, specifically focusing on RGB intensity color readings. The obtained values were subsequently printed to the serial interface in the form of a tuple comprising seven elements. The first three elements represented the RGB values as recorded by the sensor, while the subsequent three elements were the RGB ratios, determined by dividing the color value by the sum of all values. The final element encompassed the class label, denoting either Pikachu or Bulbasaur, and was pre-determined within the script. To capture a comprehensive understanding of the data, readings were collected in two formats, encompassing both raw values and ratios. Subsequently, the Serial Monitor, serving as an equivalent to standard output, provided the means to observe and record the printed values. These recorded values were then diligently transferred to a CSV file Fig. 16, facilitating subsequent analysis and training procedures.

```
Red,Green,Blue,RedRatio,GreenRatio,BlueRatio,Class
76,79,58,0.357,0.371,0.272,0
81,82,60,0.363,0.368,0.269,0
85,84,59,0.373,0.368,0.259,0
86,85,59,0.374,0.370,0.257,0
86,85,57,0.377,0.373,0.250,0
84,82,57,0.377,0.368,0.256,0
83,80,55,0.381,0.367,0.252,0
83,78,54,0.386,0.363,0.251,0
83,75,55,0.390,0.352,0.258,0
84,75,55,0.393,0.350,0.257,0
86,75,56,0.396,0.346,0.258,0
87,73,55,0.405,0.340,0.256,0
87,69,53,0.416,0.330,0.254,0
87,67,53,0.420,0.324,0.256,0
89,68,54,0.422,0.322,0.256,0
90,68,54,0.425,0.321,0.255,0
92,68,54,0.430,0.318,0.252,0
94,68,55,0.433,0.313,0.253,0
97,67,54,0.445,0.307,0.248,0
100,67,54,0.452,0.303,0.244,0
104,73,57,0.444,0.312,0.244,0
106,80,60,0.431,0.325,0.244,0
106,88,63,0.412,0.342,0.245,0
```

Figure 16. Recorded values

To view the result, the describe method was used. The describe method is a function available in the pandas[JAM+22] library for the DataFrame[PMX+20] object. It provides a summary of the

descriptive statistics for the columns in a DataFrame [PMX+20]. By default, it provides a summary for the numeric columns, including the count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum values. The statistical characteristics can be seen in Fig. 17.

If the DataFrame contains numerical data, the description contains this information for each column:

- count - The number of not-empty values;
- mean - The average (mean) value;
- std - The standard deviation;
- min - the minimum value;
- 25% - The 25% percentile;
- 50% - The 50% percentile;
- 75% - The 75% percentile;
- max - the maximum value.

Out[34]:

	Red	Green	Blue	RedRatio	GreenRatio	BlueRatio	Class
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.0
mean	46.623500	46.858500	32.379000	0.370595	0.368914	0.260478	0.0
std	31.925199	31.976424	21.055332	0.031909	0.017359	0.027647	0.0
min	4.000000	4.000000	3.000000	0.208000	0.267000	0.192000	0.0
25%	22.000000	22.000000	16.000000	0.355000	0.362000	0.243000	0.0
50%	38.000000	38.000000	27.000000	0.371000	0.371000	0.257000	0.0
75%	65.000000	67.000000	47.000000	0.389000	0.379000	0.273000	0.0
max	162.000000	159.000000	106.000000	0.533000	0.417000	0.409000	0.0

In [35]:
```
bulbasaur.describe()
```

Out[35]:

	Red	Green	Blue	RedRatio	GreenRatio	BlueRatio	Class
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.0
mean	18.196000	31.676000	30.708500	0.233561	0.396209	0.370193	1.0
std	11.829013	21.067177	22.608762	0.025913	0.033080	0.038896	0.0
min	3.000000	5.000000	3.000000	0.133000	0.295000	0.250000	1.0
25%	9.000000	14.000000	13.000000	0.216000	0.371000	0.333000	1.0
50%	15.000000	27.000000	25.000000	0.228000	0.385000	0.375000	1.0
75%	25.000000	44.000000	43.000000	0.247000	0.424000	0.404000	1.0
max	72.000000	131.000000	141.000000	0.338000	0.478000	0.477000	1.0

Act
Go t

Figure 17. Statystical characterstics

3.1.2.2. Object detection result based on different classical methods

The steps involved in deriving the results are as follows:

1. Data Loading and Preparation:

   Two CSV files, *'pikachu_complete.csv'* and *'bulbasaur_complete.csv'*, are loaded into two separate pandas DataFrame objects, pikachu and bulbasaur. Each of these files contains color data (RGB values and their respective ratios) related to images of the two Pokémon characters. To control the size of the dataset and ensure consistency in the analysis, a sample of 2000 instances is drawn from each DataFrame using a predefined random state.

2. Data Scaling:

   The Red, Green, and Blue (RGB) color values of the data for each character are normalized to a range of 0 to 255 using the MinMaxScaler from sklearn.preprocessing. The resulting scaled data are stored in *pikachu_vals_scaled* and *bulbasaur_vals_scaled*.

3. Data Visualization:

   Two functions, *draw_colors_histogram* and *draw_3d_plot*, are defined to visualize the color distributions in 2D and 3D, respectively. These functions are, however, not invoked in the provided code snippet.

4. Data Concatenation and Splitting:

   The color ratios (RedRatio, GreenRatio, and BlueRatio) and their respective class labels (Class) for both Pikachu and Bulbasaur are concatenated into a single DataFrame X. The class labels are separated from the predictors to form the target variable y. Then, the X and y datasets are split into training and testing subsets, with 70% of the data used for training and the remaining 30% for testing.

5. Model Training and Evaluation:

   A Logistic Regression model is instantiated and evaluated using 5-fold cross-validation on the training data (*X_train* and *y_train*). This gives an initial estimate of the model performance. The same Logistic Regression model is then fitted to the training data.

6. Model Prediction and Performance Reporting:

   The trained model is used to make predictions on the testing data (*X_test*). The results of these predictions are compared with the actual labels (*y_test*) to generate a classification report, which provides a comprehensive overview of model performance. This includes metrics such as precision, recall, f1-score, and support for each class, as well as overall accuracy.

In the final step, a threshold of 0.50 is applied to the predicted probabilities (*y_pred*) to convert them into class predictions. Based on *y_pred* and *y_test* MSE as well as RMSE were calculated.

Table 4. Results of different classical methods

Algorithms	Accuracy	Precision	Recall	F1	Class
Linear regression	0.98	0.98	0.97	0.98	0
		0.97	0.98	0.98	1
ARDG Regression	0.98	0.98	0.97	0.98	0
		0.97	0.98	0.98	1
Bayesian ridge	0.98	0.98	0.97	0.98	0
		0.97	0.98	0.98	1
Dummy regression	0.50	0.00	0.00	0.00	0
		0.50	1.00	0.67	1
Lasso	0.50	0.00	0.00	0.00	0
		0.50	1.00	0.67	1
Lasso LARS	0.98	0.98	0.97	0.98	0
		0.97	0.98	0.98	1
Passive aggressive regression	0.63	1.00	0.26	0.41	0
		0.57	1.00	0.73	1
Perceptron	0.97	1.00	0.94	0.97	0
		0.94	1.00	0.97	1
RANSAC regression	0.50	0.00	0.00	0.00	0
		0.50	1.00	0.67	1
SGD regressor	0.98	0.99	0.97	0.98	0
		0.97	0.99	0.98	1
Theil regression	0.98	0.98	0.97	0.98	0
		0.97	0.98	0.98	1
Tweedie regression	0.98	0.99	0.96	0.98	0
		0.96	0.99	0.98	1

Let's compare measure square error(MSE) and root measure square error(RMSE)

Table 5. Compare MSE and RMSE

Algorithms	MSE	RMSE
Linear regression	0.02083	0.14433
ARDG Regression	0.03745	0.19354
Bayesian ridge	0.03745	0.19354
Dummy regression	0.25000	0.50000
Lasso	0.25000	0.50000
Lasso LARS	0.05620	0.23706
Passive aggressive regression	0.33377	0.57773
Perceptron	0.03333	0.18257
RANSAC regression	0.50083	0.70769
SGD regressor	0.09456	0.30751
Theil regression	0.05690	0.23854
Tweedie regression	0.24612	0.49610

Based on these results, it appears that linear regression has the lowest mean squared error (MSE) and root mean squared error (RMSE) among the algorithms tested. MSE (Mean Squared Error) and

RMSE (Root Mean Squared Error) are both metrics used to measure the performance of regression models. MSE is the average squared difference between the predicted values and the actual values in a dataset [Zac21]. RMSE is the square root of MSE [Zac21].

One of the main differences between these two metrics is that RMSE is measured in the same units as the response variable, while MSE is measured in squared units of the response variable [Zac21]. This makes RMSE easier to interpret because it provides an estimate of the average deviation between predicted and actual values in terms of the unit being predicted [All22].

Both measures are necessary because they provide different information about model performance. MSE is more sensitive to large errors because it squares the error, which can be useful when working on models where occasional large errors must be minimized [All22]. RMSE, on the other hand, provides an estimate of the average deviation between predicted and actual values that is easier to interpret and understand for end users [All22]. This suggests that it is the most accurate model for the given data. The ARDG Regression, Bayesian ridge, and Theil regression algorithms also had relatively low MSE and RMSE values, indicating that they may also be effective for this task. On the other hand, the Dummy regression, Lasso, Passive aggressive regression, RANSAC regression, and Tweedie regression algorithms had relatively high MSE and RMSE values, indicating that they may not be as effective for this task. The algorithms like Dummy regression, Lasso, Passive aggressive regression, RANSAC regression, and Tweedie regression are all regression algorithms, which are designed to predict a continuous output. They are trying to predict a continuous output (like predicting house prices, temperature, etc.) rather than a discrete class label. That could be the main reason why they show higher MSE (Mean Squared Error) and RMSE (Root Mean Squared Error) values compared to the other algorithms.

Let's examine each of these models:

1. Dummy Regression: This algorithm creates a model that simply predicts the mean (or another constant value) regardless of the input. It's a simple baseline model that is not expected to perform well, as it doesn't consider the input features.

2. Lasso: Lasso (Least Absolute Shrinkage and Selection Operator) is a regression analysis method that performs both variable selection and regularization to enhance prediction accuracy and interpretability. It may not perform well if there are highly correlated predictors because it arbitrarily selects any one predictor and does zero coefficient for all others.

3. Passive Aggressive Regression: This algorithm is more suitable for large-scale learning. It is not a good choice for a small dataset as it might lead to inaccurate predictions.

4. RANSAC Regression: This algorithm is robust to outliers, but if the dataset doesn't contain significant outliers, the algorithm might perform poorly.

5. Tweedie Regression: It is a generalization of several machine learning models but may not be suitable for this specific problem.

Overall, it seems that linear regression and some of the other algorithms tested could be good options for this particular dataset.

In summary, the results of the experiment suggest that linear regression and some of the other algorithms tested may be effective for the given data. However, it is important to consider the characteristics of the data, the specific parameters used, and the complexity of the algorithms when selecting one for a particular task. Evaluating an algorithm using multiple datasets and a range of different conditions can help to get a more comprehensive understanding of its capabilities.

3.1.3 Summary

This text describes a machine learning project that uses a logistic regression model to classify Pokemon objects as Pikachu or Bulbasaur based on their RGB values. In the project described in the text, a logistic regression model is used to classify Pokemon objects as Pikachu or Bulbasaur based on their RGB values. While it may be possible to classify these objects by simply counting the number of yellow pixels in an image, this approach may not always provide accurate results. For example, if the lighting conditions are poor or if the colors in the image are similar to one another, counting yellow pixels may not accurately distinguish between Pikachu and Bulbasaur. Additionally, this approach may not be able to handle more complex classification tasks, such as distinguishing between multiple Pokemon species. Using a machine learning algorithm such as logistic regression allows the model to analyze multiple features of the image and make more accurate predictions about the class of the object. The model can be trained on a large dataset to improve its accuracy over time. The model is trained in TensorFlow and then converted to TensorFlow Lite to be used on an Arduino device. The project involves collecting training data by scanning objects with a color sensor and printing the resulting RGB values to a CSV file. The RGB values are then upscaled using scikit-learn's MinMaxScale and visualized using a *draw_colors function*. The text also mentions other machine learning algorithms that have been used or could potentially be used for object detection, including ARDRegression and Bayesian ridge.

TensorFlow Lite is a lightweight machine learning framework designed to run on edge devices with limited computational resources, such as smartphones, microcontrollers, and other embedded systems. However, TensorFlow Lite does not work with the Arduino Uno for several reasons:

1. Memory and storage limitations: The Arduino Uno has only 2KB of SRAM and 32KB of flash memory, which are insufficient for running even the simplest TensorFlow Lite models. Neural networks typically require a significant amount of memory for storing model weights and intermediate computations.

2. Processing power: The Arduino Uno is based on an 8-bit ATmega328P microcontroller running at 16 MHz, which lacks the processing power required to run TensorFlow Lite models efficiently. Deep learning models are computationally demanding and require more powerful processors, such as ARM Cortex-M series or specialized hardware accelerators, for efficient execution.

3. Unsupported architecture: TensorFlow Lite primarily targets ARM Cortex-M series microcontrollers and other more powerful processors, like those based on the ARM Cortex-A se-

ries. The ATmega328P used in the Arduino Uno has a different instruction set and architecture, which is not directly supported by TensorFlow Lite.

4. Limited library support: The Arduino Uno lacks the necessary libraries and support for running TensorFlow Lite. The framework is built in C++ and requires a certain level of standard library support, which is not available on the Arduino Uno platform.

3.1.4 Using computer vision library for detecting handwritten digits

The Modified National Institute of Standards and Technology (MNIST) dataset is a widely utilized resource in the realm of machine learning, particularly for computer vision applications. This dataset, comprising 60,000 training images and 10,000 test images of handwritten digits from 0 to 9, each sized at 28x28 pixels, is renowned for its relatively small size, structural coherence, and ease of use. Its popularity is further accentuated by the breadth of research and numerous baseline results available for comparative analysis. The versatility of the MNIST dataset has been demonstrated by its application across an array of model types, including neural networks, support vector machines, and decision trees. As such, it frequently serves as a benchmark in evaluating the performance of diverse machine learning algorithms.

Moreover, variants of the MNIST dataset have been curated for specific needs. For instance, the Extended MNIST (EMNIST) dataset includes additional images of letters and digits in both uppercase and lowercase formats. The Fashion-MNIST dataset incorporates images of fashion products, such as clothing and accessories.

The *EloquentTinyML* library presents a viable pathway for integrating the MNIST dataset with microcontroller deployments. In a typical use-case scenario, a Convolutional Neural Network (CNN) is trained on the MNIST dataset using Tensorflow, following which the model is deployed to an Arduino board via the *EloquentTinyML* library. This library facilitates the deployment of Tensorflow Lite for Microcontrollers models to Arduino boards using the Arduino Integrated Development Environment (IDE). A tool named *tinymlgen* may be utilized to export the Tensorflow Lite model into a C array, which is primed for loading by the *EloquentTinyML* library. This intricate orchestration of tools and datasets exemplifies the potential of harnessing machine learning capabilities in microcontroller environments.

3.1.4.1. Dataset

In this project, the well-known digits dataset was utilized provided by *scikit-learn*. The data was loaded and standardized by dividing each feature by the maximum feature value, effectively scaling the feature values between 0 and 1.

In the code below can be seen how it was done:

```
from sklearn.datasets import load_digits

def get_data():
    x_values, y_values = load_digits(return_X_y=True)
```

```
    x_values /= x_values.max()
    ...
    return x_train, x_test, x_validate, y_train, y_test,
        y_validate
```

As can be seen, the dataset which was used was taken *from sklearn.datasets import load_digits*. *load_digits* is a function that loads and returns the digits dataset (classification). This dataset is a copy of the test set of the UCI ML hand-written digits datasets. These digits in the dataset have been size-normalized and centered in a fixed-size image (8x8 pixels) with values (the grey-scale pixel values) ranging from 0 to 16.

The function *load_digits(n_class=10)* will return a dictionary-like object with the following attributes:

- data: (*n_samples*, *n_features*) array where *n_samples* is the number of samples and *n_features* is the number of features. Each feature corresponds to one pixel in the 8x8 image, and the feature value is the grey-scale pixel intensity.
- target: (*n_samples*,) array of target values, which are the digits themselves (from 0 to 9). target_names: array of shape (*n_class*,) giving the digit for each class.
- images: (*n_samples, 8, 8*) array of the original images. DESCR: a string giving a detailed description of the dataset.

This function is often used in machine learning tutorials as the digits dataset is small and doesn't require any data preprocessing, making it suitable for learning purposes.

3.1.4.2. Experimental Part

Model Construction and Training

For the experimental process, the TensorFlow library was used to construct and train a Convolutional Neural Network (CNN). The network was kept simple, consisting of a single convolutional layer followed by the output layer. The Adam optimizer was employed along with the *SparseCategoricalCrossentropy* loss function. The network was trained for 50 epochs using a batch size of 16.

```
# create a CNN
model = tf.keras.Sequential()
model.add(layers.Conv2D(8, (3, 3), activation='relu', input_shape
    =(8, 8, 1)))
model.add(layers.Flatten())
model.add(layers.Dense(len(np.unique(y_train))))

model.compile(optimizer='adam', loss=tf.keras.losses.
    SparseCategoricalCrossentropy(from_logits=True), metrics=['
    accuracy'])
```

```
model.fit(x_train, y_train, epochs=50, batch_size=16,
    validation_data=(x_validate, y_validate))
return model, x_test, y_test
```

Model Evaluation

The model's performance was evaluated by predicting the test set and calculating the accuracy of the predictions. The model achieved an accuracy of approximately 97% (Figs. 18 to 21).

```
def test_model(model, x_test, y_test):
    x_test = (x_test / x_test.max()).reshape((len(x_test), 8, 8,
        1))
    y_pred = model.predict(x_test).argmax(axis=1)

    print('ACCURACY', (y_pred == y_test).sum() / len(y_test))
```

Model Export

The process typically involves training the model on a more powerful machine, such as a computer, and then deploying the trained model on a microcontroller like the Arduino Nano 33 BLE Sense. The microcontroller does not perform the training, but it uses the trained model to make predictions or inferences.

The primary reasons for this approach are the computational and memory limitations of microcontrollers. Training a machine learning model can be computationally intensive and requires significant memory resources, particularly for large datasets or complex models. For this reason, the training process is typically carried out on a computer or a server, which has the necessary computational power and memory.

Once the model has been trained, it can be converted into a format suitable for use on a microcontroller. For instance, if we are using TensorFlow for training, you can convert the trained model into a TensorFlow Lite for Microcontrollers model. This version of TensorFlow is specifically designed for microcontrollers and other hardware with limited resources. It allows the trained model to run efficiently on such devices despite their constraints.

After the model has been converted, it can be loaded onto the microcontroller. The microcontroller can then use the model to make predictions based on the data it receives from its sensors. This process is often referred to as inference or prediction.

```
Epoch 38/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0245 - accuracy: 0.9972 - val_loss: 0.4973 - val_accuracy: 0.8917
Epoch 39/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0235 - accuracy: 0.9972 - val_loss: 0.4952 - val_accuracy: 0.8944
Epoch 40/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0225 - accuracy: 0.9972 - val_loss: 0.5200 - val_accuracy: 0.9000
Epoch 41/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0215 - accuracy: 0.9963 - val_loss: 0.4782 - val_accuracy: 0.9000
Epoch 42/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0205 - accuracy: 0.9963 - val_loss: 0.5067 - val_accuracy: 0.8944
Epoch 43/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0190 - accuracy: 0.9972 - val_loss: 0.4827 - val_accuracy: 0.9028
Epoch 44/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0177 - accuracy: 0.9981 - val_loss: 0.5037 - val_accuracy: 0.8972
Epoch 45/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0172 - accuracy: 0.9981 - val_loss: 0.4982 - val_accuracy: 0.9000
Epoch 46/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0164 - accuracy: 0.9991 - val_loss: 0.5091 - val_accuracy: 0.8917
Epoch 47/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0158 - accuracy: 0.9981 - val_loss: 0.5317 - val_accuracy: 0.8972
Epoch 48/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0150 - accuracy: 0.9991 - val_loss: 0.5277 - val_accuracy: 0.8972
Epoch 49/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0157 - accuracy: 0.9991 - val_loss: 0.5328 - val_accuracy: 0.8944
Epoch 50/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0146 - accuracy: 0.9981 - val_loss: 0.5074 - val_accuracy: 0.9000
12/12 [==============================] - 0s 554us/step
ACCURACY 0.9610027855153204
```

Figure 18. The result for Arduino Uno (accuracy - 0.96)

```
Epoch 42/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0251 - accuracy: 0.9972 - val_loss: 0.5028 - val_accuracy: 0.8972
Epoch 43/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0242 - accuracy: 0.9981 - val_loss: 0.4820 - val_accuracy: 0.8889
Epoch 44/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0237 - accuracy: 0.9972 - val_loss: 0.4960 - val_accuracy: 0.9056
Epoch 45/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0244 - accuracy: 0.9954 - val_loss: 0.4966 - val_accuracy: 0.8917
Epoch 46/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0208 - accuracy: 0.9991 - val_loss: 0.5214 - val_accuracy: 0.8861
Epoch 47/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0200 - accuracy: 0.9981 - val_loss: 0.4975 - val_accuracy: 0.8889
Epoch 48/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0205 - accuracy: 0.9972 - val_loss: 0.5108 - val_accuracy: 0.8972
Epoch 49/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0185 - accuracy: 0.9981 - val_loss: 0.5088 - val_accuracy: 0.8944
Epoch 50/50
68/68 [==============================] - 0s 1ms/step - loss: 0.0204 - accuracy: 0.9972 - val_loss: 0.5375 - val_accuracy: 0.8833
12/12 [==============================] - 0s 558us/step
ACCURACY 0.9498607242339833
```

Figure 19. The result for STM32 (accuracy - 0.94)

```
Epoch 45/50
 1/68 [..............................] - ETA: 0s - loss: 0.0226 - accuracy: 1.0
000
68/68 [==============================] - 0s 1ms/step - loss: 0.0178 - accu
racy: 0.9981 - val_loss: 0.4887 - val_accuracy: 0.8972
Epoch 46/50
 1/68 [..............................] - ETA: 0s - loss: 0.0381 - accuracy: 1.0
000
68/68 [==============================] - 0s 1ms/step - loss: 0.0161 - accu
racy: 0.9991 - val_loss: 0.4638 - val_accuracy: 0.9028
Epoch 47/50
 1/68 [..............................] - ETA: 0s - loss: 0.0056 - accuracy: 1.0
000
68/68 [==============================] - 0s 1ms/step - loss: 0.0157 - accu
racy: 1.0000 - val_loss: 0.4978 - val_accuracy: 0.8944
Epoch 48/50
 1/68 [..............................] - ETA: 0s - loss: 0.0026 - accuracy: 1.0
000
68/68 [==============================] - 0s 1ms/step - loss: 0.0159 - accu
racy: 0.9981 - val_loss: 0.4999 - val_accuracy: 0.9000
Epoch 49/50
 1/68 [..............................] - ETA: 0s - loss: 0.0160 - accuracy: 1.0
000
68/68 [==============================] - 0s 1ms/step - loss: 0.0148 - accu
racy: 0.9991 - val_loss: 0.4913 - val_accuracy: 0.9000
Epoch 50/50
 1/68 [..............................] - ETA: 0s - loss: 0.0326 - accuracy: 1.0
000
65/68 [============================>..] - ETA: 0s - loss: 0.0136 - accuracy
: 1.0000
66/68 [============================>.] - ETA: 0s - loss: 0.0136 - acc
uracy: 1.0000
68/68 [==============================] - 0s 2ms/step - loss: 0.0
137 - accuracy: 1.0000 - val_loss: 0.5103 - val_accuracy: 0.8972
 1/12 [=>............................] - ETA: 0s
12/12 [==============================] - 0s 4ms/step
ACCURACY 0.9610027855153204
```

Figure 20. The result for Arduino Nano 33 BLE Sense (accuracy - 0.96)

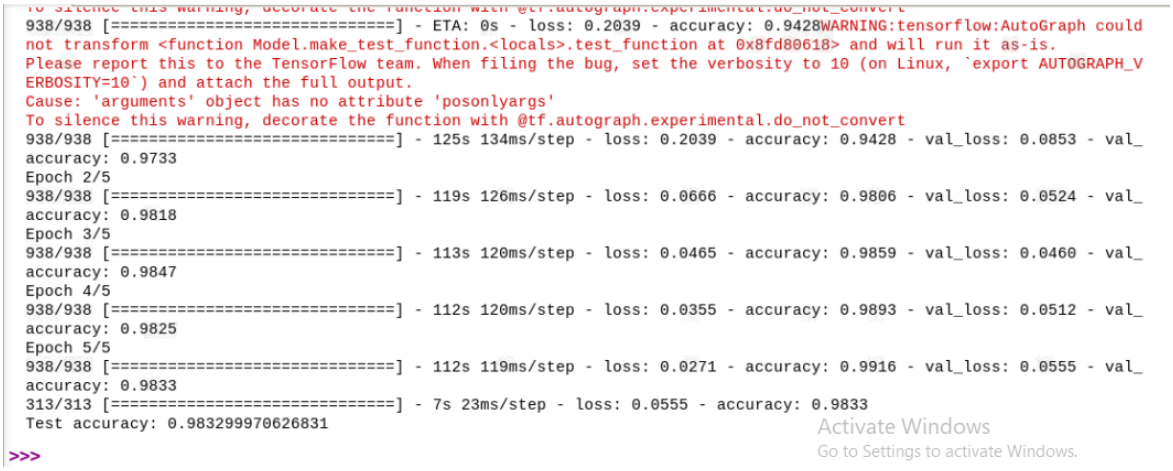


Figure 21. The result for Raspberry Pi (accuracy - 0.98)

3.1.4.3. Another library for detecting hand-written digits using the TinyMaix

Sipeed, a tech company, has made a software called *TinyMaix*. This software is impressive because it can run a model that recognizes handwritten digits, similar to the MNIST model, on a tiny computer chip called Microchip ATmega328. This chip is quite small and doesn't have much memory, so TinyMaix has been designed to use as little memory as possible. In fact, it only needs 400 lines of the main program code to work.

TinyMaix is like a small library for machine learning on tiny chips, known as microcontrollers. It doesn't have a lot of advanced features to keep things simple. For instance, it doesn't use CMSIS-NN, a group of functions that help run neural network models on Arm Cortex-M processors. Instead, it keeps things small and only needs five files to run.

The software can work in two ways of handling numbers: *INT8* and *FP32* modes. *INT8* refers to integers (whole numbers), while *FP32* refers to floating-point numbers (numbers with decimal points). Both are ways of storing data in memory. It also uses a set of tools (called SIMD, NEON, and MVEI) provided by Arm, a major chipmaker, to speed up calculations. Furthermore, it's compatible with two architectures (or designs) for RISC processors, which are chips that use a specific type of computer instruction set.

Sipeed has plans to add more features, like the *INT16* mode (which is another way of storing whole numbers), MobileNetV2 support (a model used for image recognition), and Winograd convolution optimization (a mathematical trick to make calculations quicker).

In conducting this experiment, the Conv2D algorithm was explicitly leveraged as part of the *TinyMaix* library. This algorithm is integral to the convolutional neural network (CNN) model implementation in *TinyMaix*. As a mathematical operation that applies filters to input data, Conv2D is key to tasks in machine learning and particularly in image processing tasks.

Conv2D plays a crucial role in feature extraction within images, scanning the input data for spatial hierarchies or patterns, an attribute that makes it an optimal choice when dealing with image recognition tasks. Within the scope of the *TinyMaix* library, the integration of Conv2D demonstrates the library's capabilities not only to host machine learning models but also to execute more complex operations inherent in deep learning algorithms.

Furthermore, this utilization of Conv2D within the constrained environment of *TinyMaix* further underscores the efficiency and applicability of the library. The successful implementation and execution of Conv2D within this environment, despite its computational demands, attest to the robustness and adaptability of *TinyMaix* when faced with advanced algorithmic procedures.

Thus, the integration of the Conv2D algorithm within the *TinyMai*x library forms a significant part of this experimental exploration, providing valuable insights into the software's performance, scalability, and flexibility in handling deep learning operations within a microcontroller context.

```
19:27:57.887 -> mnist demo
19:27:57.887 -> 00000000000000000000000000000000
19:27:57.887 -> 00000000000000000000000000000000
19:27:57.887 -> 00000000000000000000000000000000
19:27:57.887 -> 0000000000077AFF9500000000000
19:27:57.887 -> 000000000AFFFFFFD10000000000
19:27:57.887 -> 00000000AFFFD8BFF70000000000
19:27:57.887 -> 00000003FFD2000CF80000000000
19:27:57.887 -> 00000004FD10007FF40000000000
19:27:57.887 -> 00000000110000DFF40000000000
19:27:57.887 -> 00000000000007FFC00000000000
19:27:57.887 -> 0000000000004FFE300000000000
19:27:57.887 -> 0000000000008FF9000000000000
19:27:57.934 -> 00000000000BFF90000000000000
19:27:57.934 -> 00000000001EFE20000000000000
19:27:57.934 -> 0000000000CFF8000000000000000
19:27:57.934 -> 0000000004FFB0000000000000000
19:27:57.934 -> 000000001CFF8000000000000000
19:27:57.934 -> 000000008FFA0000000000000000
19:27:57.934 -> 00000000FFF10000000000000000
19:27:57.934 -> 00000000FFF21111000112999900
19:27:57.934 -> 00000000FFFFFFFFA8AFFFFFF70
19:27:57.934 -> 00000000AFFFFFFFFFFFFFA7730
19:27:57.934 -> 0000000007777AFFF97720000000
19:27:57.934 -> 00000000000000000000000000000000
19:27:57.934 -> 00000000000000000000000000000000
19:27:57.934 -> 00000000000000000000000000000000
19:27:57.934 -> 00000000000000000000000000000000
19:27:57.934 -> 00000000000000000000000000000000
19:27:57.988 -> ===use 49912us
19:27:57.988 -> 0: 0
19:27:57.988 -> 1: 0
19:27:57.988 -> 2: 89
19:27:57.988 -> 3: 0
19:27:57.988 -> 4: 1
19:27:57.988 -> 5: 6
19:27:57.988 -> 6: 1
19:27:57.988 -> 7: 0
19:27:57.988 -> 8: 0
19:27:57.988 -> 9: 0
19:27:57.988 -> ### Predict output is: Number 2, prob=89
```

Figure 22. The result in Arduino Uno probability is 89% (same result was in STM32 Nucleo-L0538 and Arduino Nano 33 BLE Sense)

3.1.5 Iris classification

The Iris dataset serves as a fundamental resource in the study of machine learning principles, detailing three species of the Iris flower with attributes such as petal width, petal height, sepal length, and sepal height. Typically, the classification of Iris flowers with an Arduino is achieved using the RandomForestClassifier algorithm from the sklearn library. To achieve this, it is necessary to import all relevant libraries and load the data into a pandas DataFrame. Subsequently, the dataset can be divided into training and test sets. The RandomForestClassifier can then be fitted to the training data, and its performance evaluated on the testing data using metrics such as accuracy, precision, and recall.

Once the model has been selected, C++ code can be generated using *micromlgen*, a tool that transforms machine learning models into portable C code. The code can then be loaded onto an Arduino device, thereby enabling the device to classify Iris species based on input feature data.

While the Iris dataset itself is not directly related to computer vision as it's composed of nu-

merical data rather than images, the principles used in its classification can be related to computer vision tasks. The classification of Iris species based on feature data is similar to the classification of images based on pixel data in computer vision. Just as the *RandomForestClassifier* can be trained to identify the species of an Iris based on measurements, a Convolutional Neural Network (CNN), for instance, can be trained to recognize objects in an image based on pixel patterns. Furthermore, the transformation of the trained model into C++ code for deployment on an Arduino can be mirrored in computer vision tasks, making it possible to run image recognition models on small, low-powered devices.

The Iris dataset describes 3 species of the Iris flower in terms of

- petal width
- petal height
- sepal length
- sepal height

Experiment

The experiment can be broken down into several steps:

- Image Preprocessing: With OpenCV, convert the images to grayscale and apply image processing techniques such as edge detection or segmentation to isolate the flowers in the images. Extract features (e.g., size, shape, texture) from the images corresponding to the Iris dataset's sepal and petal measurements.
- Feature Extraction: Next, we need to map these extracted features to the measurements provided in the Iris dataset. For instance, the size and shape features from the images could correspond to the petal and sepal measurements.
- Model Training: Using the *RandomForestClassifier* from the *sklearn* library, train a model with the mapped features as the predictors and the Iris species as the target. This model can then classify Iris species based on the features extracted from the images.
- Model Porting: Convert the trained *RandomForestClassifier* model into C++ code using the *micromlgen* library. This code can be incorporated into an Arduino script, allowing the Arduino device to perform the classification task.
- Arduino Integration: With the model ported to the Arduino, set up the device to capture images of Iris flowers using a camera module, preprocess these images, extract features, and classify the Iris species in real-time.

The result can be seen below fig. 23

```
Predicted class label: setosa
Predicted class: 1
Predicted class label: versicolor
Predicted class: 2
Predicted class label: virginica
```

Figure 23. The result on all controllers

3.2 Use of OpenCV library

3.2.1 Detecting color

While simple methods such as calculating R/G and G/B ratios or mapping colors to a color wheel can be effective for detecting colors in some situations, they may not always provide accurate results. For example, these methods may not work well in challenging conditions such as poor lighting or when the colors in the image are similar to one another.

Machine Learning (ML) algorithms can provide more robust color detection results by analyzing multiple features of an image and applying advanced mathematical models to accurately identify the desired color. These algorithms can be trained on large datasets to improve their accuracy over time.

For example, an ML algorithm could be trained to recognize different shades of a particular color, even when the lighting conditions vary. This would allow the algorithm to accurately detect the desired color in a wide range of situations.

The implementation is designed to identify the existence of particular colors (red, yellow, orange, blue, and green) within a video stream acquired via a webcam. This process leverages the OpenCV library, a renowned computer vision framework extensively employed for image and video processing tasks.

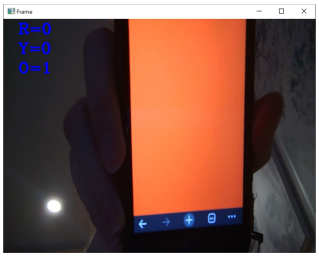
The algorithm used for color detection is a classical computer vision approach that involves converting the RGB image captured by the webcam into the HSV color space. Then, upper and lower bounds are defined for each of the five colors of interest in the HSV color space. These bounds are used to threshold the HSV image to create a binary mask for each color, which identifies the pixels that belong to the respective color range.

After the binary mask is generated, it is dilated using a 5x5 kernel, which expands the pixel regions and makes the detection more robust. Then, a bitwise AND operation is performed between the dilated mask and the original image to extract the regions of interest (i.e., the color objects) from the video frame.

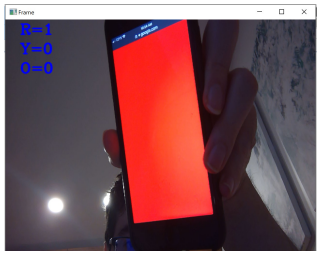
The contours of the color objects are then extracted using the *findContours()* function provided by OpenCV. These contours are used to determine whether a color is present in the video frame or not. If the area of the contour is greater than 300 pixels, the color is considered to be present, and the corresponding flag variable is set to 1. Otherwise, the flag variable is set to 0.

Finally, the flag variables are used to update the dictionary 'data', which stores the presence of each color. The *putText()* function is used to display the presence of each color as a string in the video frame.

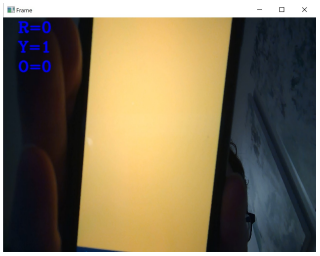
In summary, the provided code utilizes classical computer vision techniques, implemented on the Arduino Uno, to detect specific colors in a video stream captured by a webcam. OpenCV library functions are employed for thresholding, color object extraction, and contour computation. While the algorithm is relatively simple, it proves effective in detecting the desired colors within the video stream.



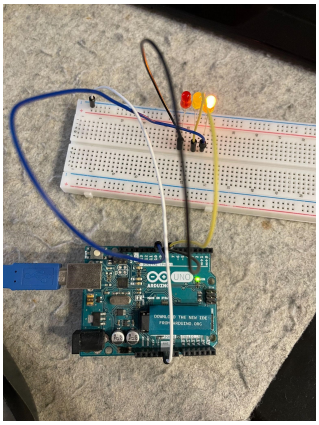
(a) Orange color



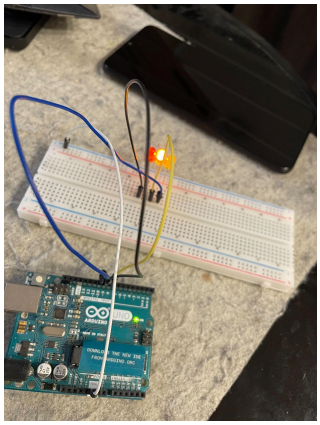
(b) Red color



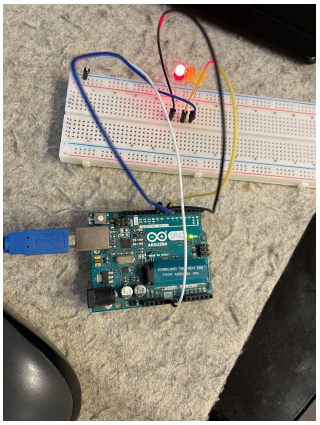
(c) Yellow color



(d) Orange color detection



(e) Red color detection



(f) Yellow color detection

Figure 24. The result of detecting color

3.2.2 Object detecting

An implemented solution captures real-time video from a webcam, conducting color detection on the acquired frames. By employing computer vision techniques and leveraging the OpenCV library, the system processes the video stream and analyzes the color information, enabling the identification of yellow and green color objects within the video stream.

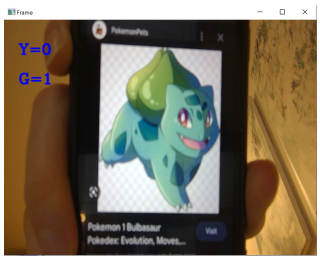
To start, necessary libraries such as *numpy*, *cv2*, and a custom 'ledLight' module are imported. The webcam is initiated using *cv2.VideoCapture()*, and variables 'c_yellow' and 'c_green' are set to zero. A 'data' dictionary is created to store detection results. The program continuously reads video frames and performs color detection within a while loop. It checks if a frame is the last in the stream, resetting the frame count to zero if so.

The current frame is read using the *webcam.read()* function, and its size is adjusted to 640x480 using the *cv2.resize()* function. The *cv2.cvtColor()* function is called to convert the color space of the frame from BGR to HSV. The program defines the lower and upper color boundaries for the yellow and green color objects using numpy arrays. The *cv2.inRange()* function is used to create a binary mask for each color range based on the HSV color space values of the frame.

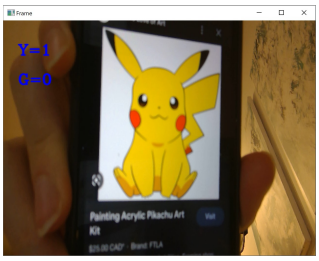
A 5x5 kernel is defined using numpy, and the *cv2.dilate()* function is called to perform morphological dilation on the binary masks. The *cv2.bitwise_and()* function is used to apply the masks to the original frame to extract the yellow and green color objects. The *cv2.findContours()* function is called to identify the contours of the color objects in the binary masks. The contours are filtered based on their area, and if the area is greater than a threshold value of 300, the color detection variables 'c_yellow' and 'c_green' are set to 1; otherwise, they are set to 0.

The 'data' dictionary is updated with the latest detection results, and the detection results are displayed on the video stream using the *cv2.putText()* function. The 'ledLight' function is called to control external LED lights based on the detection results. The program also displays the detection results in the console for debugging purposes.

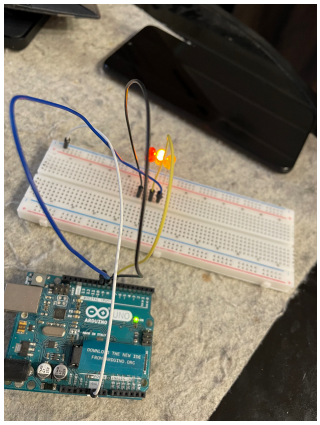
Finally, the program displays the video stream in a window using the *cv2.imshow()* function, and it waits for the user to press the 'q' key to exit the loop. Once the loop is exited, the program releases the webcam and closes all windows using the *cv2.release()* and *cv2.destroyAllWindows()* functions, respectively.



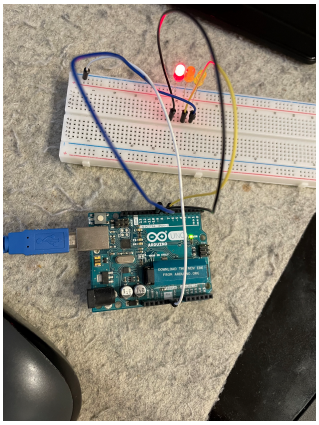
(a) Bulbasaur



(b) Pikachu



(c) Bulbasaur detection



(d) Pikachu detection

3.2.3 Control LED

Controlling an LED using MediaPipe on Arduino refers to a project that uses MediaPipe, a computer vision framework, and Arduino, a microcontroller platform, to control the behavior of an LED light. MediaPipe can be used to track hand movements and gestures, which can then be used to control the LED light.

Computer vision is a field of study that focuses on enabling computers to interpret and understand visual information from the world. MediaPipe is a framework that provides tools for building computer vision applications. In this context, MediaPipe is being used to track hand movements

and gestures, which are then used as inputs to control the behavior of an LED light connected to an Arduino board

3.2.3.1. MediaPipe framework

MediaPipe is a framework for building machine learning pipelines for processing time-series data like video, audio, etc. This cross-platform Framework works on Desktop/Server, Android, iOS, and embedded devices like Raspberry Pi and Jetson Nano.

With MediaPipe, a perception pipeline can be built as a graph of modular components, including model inference, media processing algorithms and data transformations, etc. Sensory data such as audio and video streams enter the graph, and perceived descriptions such as object-localization and face landmark streams exit the graph[LTN+19]. MediaPipe is primarily used for quickly creating perception pipelines with inference models and other components, as well as deploying these pipelines into demos and applications on various hardware platforms. It is useful for prototyping and implementing perception technology.

The MediaPipe perception pipeline is called a Graph. A pipeline is defined as a directed graph of components where each component is a Calculator[LTN+19]. In a data-flow graph, the calculators are connected by streams. Each stream represents a series of data packets over time. The packets are organized within the series based on their timestamps as they flow through the graph. The calculators and streams together form the data-flow graph.

The basic data unit in MediaPipe is a Packet. A packet consists of a numeric timestamp and a shared pointer to an immutable payload[LTN+19]. Packets are value classes that can be efficiently copied. Each copy shares ownership of the payload through reference-counting semantics and has its own timestamp.

Each node in the graph is connected to another node through a Stream. In MediaPipe, a stream refers to the edges of a MediaPipe graph that carry a sequence of packets with ascending timestamps [Kuk22]. MediaPipe calculator graphs are often used to process streams of video or audio frames for interactive applications [Goo23]. The MediaPipe framework requires that successive packets be assigned monotonically increasing timestamps [Goo23]. A stream carries a sequence of packets whose timestamps must be monotonically increasing[LTN+19]. An output stream can be connected to any number of input streams of the same type[LTN+19].

Each node in the graph is implemented as a Calculator[LTN+19]. When initializing a calculator, it declares the packet payload type that will traverse the port. Every time a graph runs, the Framework implements Open, Process, and Close methods in the calculators. Open initiates the calculator; the process repeatedly runs when a packet enters. The process is closed after an entire graph run. There exists 4 types of Calculator such as:

1. Pre-processing calculators are a family of image and media-processing calculators. The *ImageTransform* and *ImageToTensors* in the graph above fall in this category.

2. Inference calculators allow native integration with Tensorflow and Tensorflow Lite

for ML inference.

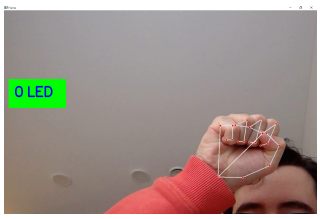
3. Post-processing calculators perform ML post-processing tasks such as detection, segmentation, and classification. *TensorToLandmark* is a post-processing calculator.

4. Utility calculators are a family of calculators performing final tasks such as image annotation.

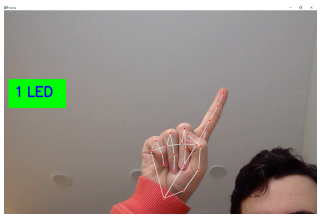
3.2.3.2. Experiment

In this task, the MediaPipe framework and OpenCV were used, and as a controller Arduino Uno was used. OpenCV was used for reading video in real time and for that *cv2.VideoCapture* function was used. *cv2.VideoCapture* - Creates a video capture object, which would help stream or display the video. After opencv was configured, mediapipe was used to recognize the number of fingers. The project does not actually involve training a machine learning model. Instead, it uses an already trained model provided by the MediaPipe library to detect hand landmarks in real-time [Goo23].

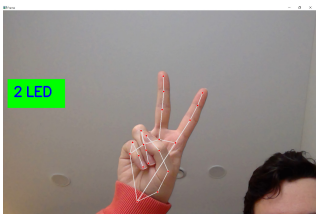
The specific model used in this project is part of the MediaPipe Hands solution. This solution employs a palm detection model Figs. 26a to 26f (a type of Convolutional Neural Network or CNN) to first identify a hand in the image, and then a separate hand landmarks model to identify the 21 individual points of a hand in the image [Goo23].



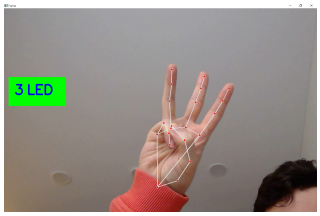
(a) All LEDs turned off



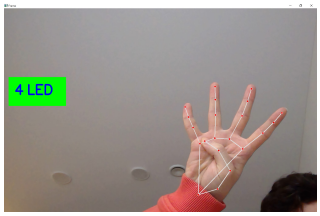
(b) 1 LED is turned on



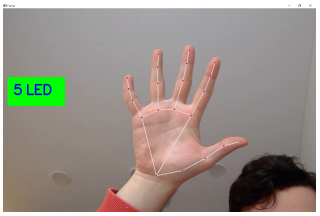
(c) 2 LEDs are turned on



(d) 3 LEDs are turned on



(e) 4 LEDs are turned on



(f) 5 LEDs are turned on

Figure 26. Recognizing number of fingers

The scheme of connections can be seen below:

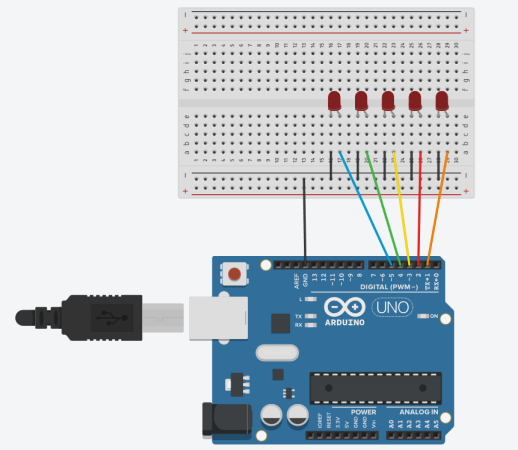


Figure 27. Scheme of connections

Results

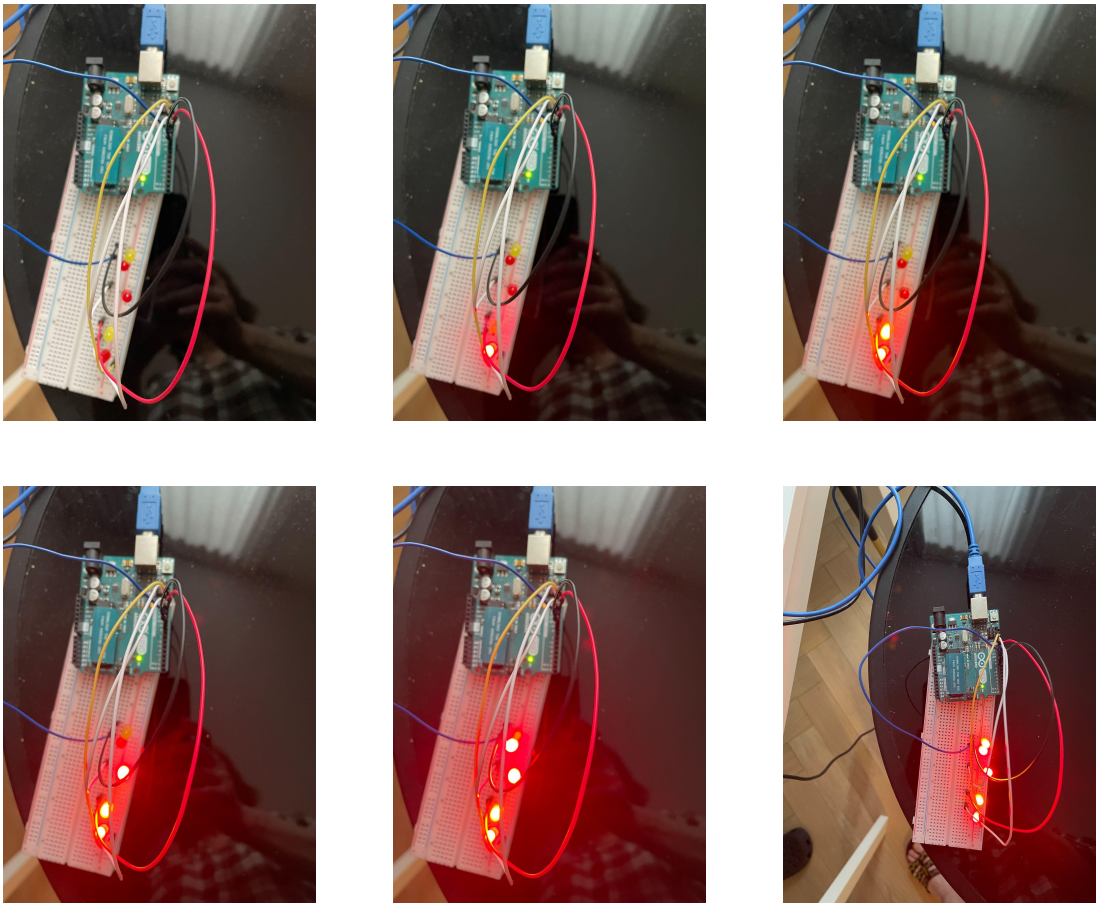


Figure 28. Results switch on/off LED lights

3.2.4 Detecting traffic signs

3.2.4.1. Motivation

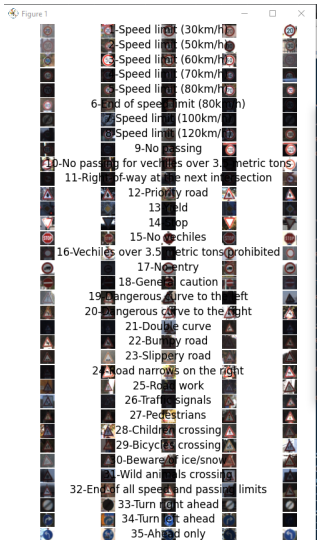
Traffic sign detection is a critical aspect of modern transportation systems, from enhancing road safety to providing essential data for autonomous vehicle technologies. However, achieving ef-

ficient and accurate traffic sign detection remains a challenging task due to factors like varying light conditions, diverse sign designs, and occlusions. In this study, a CNN-based model was developed to classify images from a custom dataset using the Keras deep learning framework [https://keras.io/].

3.2.4.2. Methodology

Dataset

The dataset, sourced from Kaggle, consists of images stored in the ”myData” folder, with each subfolder representing a distinct class. The data is meticulously partitioned into training, validation, and testing sets, with 80% dedicated for training, while the remaining 20% is equally divided between validation and testing. As part of preprocessing, the images are converted to grayscale, equalized for histogram, and normalized for pixel values.



(a) Dataset

Convolutional Neural Network Model

The CNN architecture consists of the following layers:

1. Two *Conv2D* layers with 60 filters each and a kernel size of (5,5), followed by a ReLU activation function.
2. A *MaxPooling2D* layer with a pool size of (2,2).
3. Two Conv2D layers with 30 filters each and a kernel size of (3,3), followed by a ReLU activation function.
4. A *MaxPooling2D* layer with a pool size of (2,2).
5. A *Dropout* layer with a rate of 0.5 to prevent overfitting.
6. A *Flatten* layer to convert the feature maps into a one-dimensional array.
7. A *Dense* layer with 500 nodes and a ReLU activation function.

8. A *Dropout* layer with a rate of 0.5.

9. An output *Dense* layer with *softmax* activation function for multi-class classification.

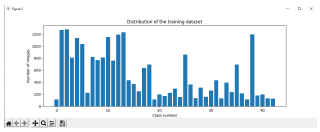
The code below as follows:

```
def myModel():
    no_Of_Filters=60
    # this is the kernel that move around the image to get the
      features.
    size_of_Filter=(5,5)
    # this would remove 2 pixels from each border when using 32
      32 image
    size_of_Filter2=(3,3)
    # scale down all feature map to gernalize more, to reduce
      overfitting
    size_of_pool=(2,2)
    # no. of nodes in hidden layers
    no_Of_Nodes = 500
    model= Sequential()
    # adding more convolution layers = less features but can
      cause accuracy to increase
    model.add((Conv2D(no_Of_Filters,size_of_Filter,input_shape=(
      imageDimesions[0],imageDimesions[1],1),activation='relu'))
      )
    model.add((Conv2D(no_Of_Filters, size_of_Filter, activation='
      relu')))
    # does not effect the depth/no of filters
    model.add(MaxPooling2D(pool_size=size_of_pool))

    model.add((Conv2D(no_Of_Filters//2, size_of_Filter2,
      activation='relu')))
    model.add((Conv2D(no_Of_Filters // 2, size_of_Filter2,
      activation='relu')))
    model.add(MaxPooling2D(pool_size=size_of_pool))
    model.add(Dropout(0.5))

    model.add(Flatten())
    model.add(Dense(no_Of_Nodes,activation='relu'))
    # inputs nodes to drop with each update 1 all 0 none
    model.add(Dropout(0.5))
    # output layer
```

```
model.add(Dense(noOfClasses,activation='softmax'))
# compile model
model.compile(Adam(lr=0.001),loss='categorical_crossentropy',
    metrics=['accuracy'])
return model
```

Data Augmentation

To increase the generalizability of the model, we employ data augmentation techniques using the *ImageDataGenerator* class from Keras. Data augmentation is used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. The idea is to replicate the kinds of variations we expect in the real-world, thereby increasing the diversity of data available for training models, without actually collecting new data. The data augmentation is performed using *ImageDataGenerator* which generates batches of tensor image data with real-time data augmentation. The data will be looped over in batches indefinitely. The applied augmentations include width and height shifts, zooming, shearing, and rotation.



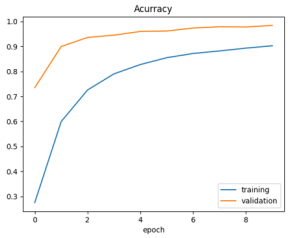
(a) Distribution of the training dataset

(b) Normalized image

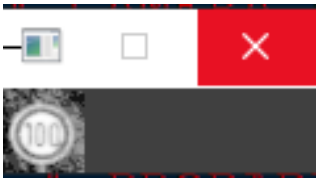
Results and Discussion

In this research, an array of hardware devices including Arduino Uno, STM32, Arduino Nano 33 BLE Sense, and Raspberry Pi were utilized in the detection process. Upon evaluation, these different hardware systems demonstrated analogous results in terms of the accuracy of traffic sign detection.

The model is trained for ten epochs with a batch size of 50 and 2,000 steps per epoch using the augmented training data. The training and validation loss and accuracy are plotted to analyze the model's performance.



(a) Validation loss and accuracy



(b) Grayscaling on the example of one of the datasets

(c) Result of detection (class 14 - stop, probability - 79.82%)

```
Send to arduio value 79.82
```

(d) Result on Arduino (probability - 79.82%)

The model achieves a high test accuracy (accuracy - 0.91, test sore - 0.81), demonstrating its capability to classify images from the custom dataset effectively. The use of data augmentation contributes to the model's generalizability, preventing overfitting and increasing its robustness to variations in input data.

Summary

This study demonstrates the effectiveness of a Convolutional Neural Network-based model combined with data augmentation for image classification tasks. The proposed method achieves high accuracy (80%) on a custom dataset, validating its utility for real-world applications. Future work could involve the exploration of more advanced CNN architectures, optimization techniques, and transfer learning to further improve the model's performance.

4 Conclusions

In this discussion, the Internet of Things (IoT) architecture was outlined, including the various components that make up an IoT system such as end devices, software, communication protocols, and platforms. The use of vision sensors in IoT, including proximity sensors, optical sensors, vision sensors, and fiber optic sensors, was also described. The difference between vision sensors and vision systems was explained. The topic of machine learning methods and IoT was then introduced, including the definitions of classical and deep learning methods and the various types of machine learning tasks and algorithms. The challenges of using machine learning in IoT were also mentioned. The use of computer vision for IoT in cloud-based solutions, such as TinyML and EdgeImpulse, as well as the use of third-party libraries, was discussed. The topic of developing computer vision in microcontrollers was also covered, including the use of sensors like CMOS and CCD image sensors and libraries like OpenCV and TensorFlow. The process of software design for computer vision in microcontrollers, including the selection of appropriate microcontrollers and

software development environments, was also described. Finally, the use of the TensorFlow library for tasks such as detecting color and object detection was discussed.

The performance of various regression algorithms was compared using a dataset, and it was found that linear regression had the lowest mean squared error and root mean squared error values, indicating that it was the most accurate model.

In conclusion, computer vision and machine learning play a significant role in the field of IoT, enabling devices to process and analyze data in order to make decisions and take actions. There are various algorithms and methods available for implementing computer vision and machine learning in IoT systems, ranging from classical methods to deep learning approaches. These methods can be applied in both cloud-based and on-device settings, depending on the specific requirements and constraints of the application. However, using machine learning in IoT systems also presents certain challenges, such as the need for large amounts of data and computing resources, as well as the complexity of some algorithms. To overcome these challenges, developers can make use of specialized tools and libraries, as well as carefully consider their choice of hardware and software platforms. By understanding the capabilities and limitations of different machine learning methods and applying them appropriately, it is possible to build powerful and intelligent IoT systems that can improve our lives and solve real-world problems.

Several experiments were observed, and these various projects and studies demonstrate the versatility and effectiveness of machine learning and computer vision techniques in solving practical problems, such as color detection, object classification, and image processing. The applications deployed on platforms like Arduino Uno, STM32, Arduino Nano 33 BLE Sense, and Raspberry Pi utilize these microcontrollers as the core processing unit for executing the programs. Microcontrollers provide a compact, cost-effective, and energy-efficient solution for implementing machine learning and computer vision tasks on embedded devices. The experiments showcased the potential of various machine learning algorithms, such as linear regression, random forest, and deep learning models, for deployment on resource-constrained platforms, enabling a wide range of applications in the field of embedded systems and the Internet of Things (IoT). The term 'various algorithms' refers to the diverse machine learning algorithms implemented and tested during the experiments. These include, but are not limited to, deep learning methods like Convolutional Neural Networks (CNN) used in the MNIST classification, classic regression methods, including linear regression for object detection, and others for the multiple IoT tasks. The number of algorithms can vary based on the specific problem at hand and the available data. Additionally, the classical computer vision techniques and the OpenCV library demonstrated their effectiveness in solving color detection and object recognition problems. While both classic computer vision techniques and deep learning models have their unique strengths and weaknesses, it is important to choose the most appropriate approach based on the problem at hand, the available data, and the computational resources. Classic computer vision techniques, such as the color detection algorithm presented in this project, can provide effective and efficient solutions for certain applications, particularly in cases where the task can be explicitly defined and solved with simple image processing techniques. On the other hand, deep learning models, such as those implemented in the microcontroller-based

and MediaPipe-based projects, can provide more robust and accurate solutions for complex and dynamic problems, particularly in cases where the available data is large and diverse. MediaPipe-based projects deliver more robust and accurate solutions due to their advanced computer vision capabilities. By leveraging the power of machine learning, they're able to achieve higher accuracy in comparison to traditional image processing techniques. The choice between classical and deep learning methods depends largely on the complexity of the problem, available data, and computational resources. Classical methods, like linear regression, can be more efficient and easier to interpret and can perform exceptionally well with smaller datasets or simpler tasks. On the other hand, deep learning methods, like Convolutional Neural Networks, have the capability to handle more complex tasks, learn from large datasets, and automatically extract features. However, they require more computational resources and data, and the model's decision-making process might not be as transparent. It's noteworthy to mention that while deep learning models achieved higher accuracy in the MNIST and traffic sign classification tasks, classical methods were still competitive in certain scenarios like object detection, showcasing their continued relevance in the field of machine learning.

Overall, these projects and studies need benchmarks for future research and development in the field of machine learning and computer vision, demonstrating the potential of these techniques for solving real-world problems and advancing the capabilities of intelligent systems and devices.

References

[AAA+22] A. M. Alrashdi, A. E. Alrashdi, A. Alghadhban, and M. A. H. Eleiwa. Optimum gssk transmission in massive mimo systems using the box-lasso decoder. https://doi.org/10.1109/ACCESS.2022.3148329, 2022.

[AAB+20] Erwin Adi, Adnan Anwar, Zubair Baig, and Sherali Zeadally. Machine learning and data analytics for the iot. https://arxiv.org/pdf/2007.04093.pdf, 2020.

[ABJ19] Dr. J. Jegathesh Amalraj, S. Banumathi, and J. Jereena John. Iot sensors and applications: a survey. *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, 8:998–1003, 2019.

[Ack20] Daniel Ackerman. System brings deep learning to "internet of things" devices. https://news.mit.edu/2020/iot-deep-learning-1113, 2020.

[AKA02] Hussein Almuallim, Shigeo Kaneda, and Yasuhiro Akiba. 3 - development and applications of decision trees. https://doi.org/10.1016/B978-012443880-4/50047-8, 2002.

[All22] Stephen Allwright. Rmse vs mse, what's the difference? https://stephenallwright.com/rmse-vs-mse/, 2022.

[AP83] Witkin A.P. Scale-space filtering. in international joint conference on artificial intelligence. https://www.ijcai.org/Proceedings/83-2/Papers/091.pdf, 1983. International Joint Conference on Artificial Intelligence, Karlsruhe, Germany, pp. 1019-1022.

[ATT18] A. George Assaf, Mike Tsionas, and Anastasios Tasiopoulos. Diagnosing and correcting the effects of multicollinearity: bayesian implications of ridge regression. https://doi.org/10.1016/j.tourman.2018.09.008, 2018.

[BBV00] J. Bruce, T. Balch, and M. Veloso. Fast and inexpensive color image segmentation for interactive robots. https://doi.org/10.1109/IROS.2000.895274, 2000. IEEE.

[BJT+10] S. Bhattacharyya, S. Jha, K. Tharakunnel, and J. C. Westland. Data mining for credit card fraud: a comparative study. http://euro.ecom.cmu.edu/resources/elibrary/epay/1-s2.0-S0167923610001326-main.pdf, 2010. Elsevier.

[BKG20] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. https://doi.org/10.48550/arXiv.2003.05991, 2020.

[BL20] Adel Bedoui and Nicole A. Lazar. Bayesian empirical likelihood for ridge and lasso regressions. https://doi.org/10.1016/j.csda.2020.106917, 2020.

[Bor14] Eleonora Borgia. The internet of things vision: key features, applications and open issues. https://doi.org/10.1016/j.comcom.2014.09.008, 2014. Elsevier.

[Bre01] Leo Breiman. Random forests. https://doi.org/10.1023/A:1010933404324, 2001. Springer.

[BS20] Khadija El Bouchefry and Rafael S. De Souza. Supervised learning model for high-dimensional and large-scale data. https://doi.org/10.1016/B978-0-12-819154-5.00023-0, 2020. Elsevier.

[CAP+12] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. A brief introduction to opencv. https://mipro-proceedings.com/sites/mipro-proceedings.com/files/upload/sp/sp_008.pdf, 2012. IEEE.

[CDM19] G. F. Contreras, H. J. Dulce-Moreno, and R. Ardila Melo. Arduino data-logger and artificial neural network to data analysis. http://dx.doi.org/10.1088/1742-6596/1386/1/012070, 2019. IOP Publishing.

[CT22] Timothy Clem and Patrick Thomson. Static analysis at github. https://doi.org/10.1145/3486594, 2022. ACM.

[Dat16] Dataray. mos vs. ccd sensors and overview. https://dataray.com/blogs/dataray-blog/cmos-vs-ccd-sensors-and-overview, 2016.

[Dat20] Dataray. What is the difference between vision sensors and vision systems? https://www.cognex.com/blogs/machine-vision/whats-the-difference-between-vision-sensors-and-vision-systems, 2020.

[DSR11] S. Gayathri Devi, K. Selvam, and Dr. S. P. Rajagopalan. An abstract to calculate big o factors of time and space complexity of machine code. ieee, 2011. http://dx.doi.org/10.1049/cp.2011.0483, 2011. IEEE.

[DT18] Yvan Duroc and Smail Tedjini. Rfid: a key technology for humanity. http://dx.doi.org/10.1016/j.crhy.2018.01.003, 2018. Elsevier.

[Dub10] E. Dubois. *The Structure and Properties of Color Spaces and the Representation of Color Images. Synthesis Lectures on Image, Video, and Multimedia Processing*. JMorgan & Claypool Publishers, 1537 Fourth Street, San Rafael, CA 94901 USA, 2010.

[Edg22] EdgeImpulse. Edgeimpulse documentation. https://docs.edgeimpulse.com/docs/faq, 2022.

[ERF97] Osuna E., Freund R., and Girosit F. Training support vector machines: an application to face detection. 10.1109/CVPR.1997.609310, 1997. Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition.

[FFG21] Laura Freijeiro-González, Manuel Febrero-Bande, and Wenceslao González-Manteiga. A critical review of lasso and its derivatives for variable selection under dependence among covariates. https://doi.org/10.1111/insr.12469, 2021. International Statisitical Review.

[Fos97] Eric R. Fossum. Cmos image sensors: electronic camera-on-a-chip. https://doi.org/10.1109/16.628824, 1997. IEEE.

[GDD+14] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. https://doi.org/10.1109/CVPR.2014.81, 2014. IEEE Conference on Computer Vision and Pattern Recognition, pp. 580–587.

[Gir15] R. Girshick. Fast r-cnn. https://doi.org/10.1109/ICCV.2015.169, 2015. IEEE International Conference on Computer Vision (ICCV), pp. 1440–1448.

[Gol16] Peter Goldsborough. A tour of tensorflow. https://arxiv.org/abs/1610.01178, 2016.

[Goo21] Google. Tensorflow model analysis architecture. https://www.tensorflow.org/tfx/model_analysis/architecture, 2021.

[Goo23] Google. Real-time streams. https://developers.google.com/mediapipe/framework/framework_concepts/realtime_streams, 2023.

[GU16] C. Gayathri and R. Umarani. Forecasting of automatic relevance determination for feature selection (fard-fs) in financial fraud detection. https://www.researchgate.net/publication/309071482_Forecasting_of_automatic_relevance_determination_for_feature_selection_FARD-FS_in_financial_fraud_detection, 2016.

[HM78] M.J. Howes and D.V. Morgan. *Charge-coupled Devices and Systems*. John Wiley & Sons, New York, 1978.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, et al. Array programming with numpy. https://doi.org/10.1038/s41586-020-2649-2, 2020. Nature.

[Hol96] G.C. Holst. Ccd arrays, cameras, and displays. https://www.worldcat.org/title/38216530, 1996. SPIE Optical Engineering Press.

[Hor14] M. Horowitz. Computing's energy problem(and what we can do about it). https://doi.org/10.1109/ISSCC.2014.6757323, 2014. ISSCC.

[Hor93] I. Horswill. Polly: a vision-based artificial agent. 10.5555/1867270.1867393, 1993. ACM, The Proceedings of the Eleventh National Conference on Artificial Intelligence.

[JAM+22] Sajib Kumar Saha Joy, Farzad Ahmed, Al Hasib Mahamud, and Nibir Chandra Mandal. An empirical studies on how the developers discussed about pandas topics. https://arxiv.org/pdf/2210.03519v1.pdf, 2022.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. Adam: a method for stochastic optimization. https://arxiv.org/abs/1412.6980, 2015. ICLR.

[Ker20] Keras. Keras api. https://keras.io/api/, 2020.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf, 2012. "In Advances in neural information processing systems, pp. 1097–1105.

[KST+21] Tomoe Kishimoto, Masahiko Saito, Junichi Tanaka, Yutaro Iiyama, Ryu Sawada, and Koji Terashi. An improvement of object detection performance using multi-step machine learnings. https://doi.org/10.48550/arXiv.2101.07571, 2021. IEEE.

[Kuk22] Kukil. Introduction to mediapipe. https://learnopencv.com/introduction-to-mediapipe/, 2022.

[KWH21] Leandro Von Krannichfeldt, Yi Wang, and Gabriela Hug. Online ensemble learning for load forecasting. https://arxiv.org/pdf/1509.02438.pdf, 2021. IEEE Transactions on Power Systems.

[Lan05] J. Landt. The history of rfid. https://doi.org/10.1109/MP.2005.1549751, 2005. IEEE.

[LBG97] L.M. Lorigo, R.A. Brooks, and W.E.L. Grimson. Visually guided obstacle avoidance in unstructured environments. https://people.csail.mit.edu/brooks/papers/final-iros.pdf, 1997. Proceedings of IROS 97, pp. 373-379.

[LKF10] Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. Convolutional networks and applications in vision. https://doi.org/10.1109/ISCAS.2010.5537907, 2010. IEEE.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. https://doi.org/10.1023/B:VISI.0000029664.99615.94, 2004. Springer.

[LPS21] Marco Lombardi, Francesco Pascale, and Domenico Santaniello. Internet of things: a general overview between architectures, protocols and applications. https://www.mdpi.com/2078-2489/12/2/87, 2021. MDPI.

[LTN+19] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, et al. Mediapipe: a framework for building perception pipelines. https://doi.org/10.48550/arXiv.1906.08172, 2019. Google Research.

[LWX+15] L.Wang, W.Ouyang, X.Wang, and H. Lu. Visual tracking with fully convolutional networks. https://openaccess.thecvf.com/content_iccv_2015/papers/Wang_Visual_Tracking_With_ICCV_2015_paper.pdf, 2015. ICCV, pp. 3119–3127.

[Mac91] David J.C. MacKay. Bayesian methods for adaptive models. https://thesis.library.caltech.edu/25/1/MacKay_djc_1992.pdf, 1991. California Institute of Technology.

[MK20] Kritika Malhotra and Yankit Kumar. Challenges to implement machine learning in embedded systems. https://doi.org/10.1109/ICACCCN51052.2020.9362893, 2020. IEEE.

[MMR91] Peter Meer, Doron Mintz, and Azriel Rosenfeld. Robust regression methods for computer vision: a review. https://doi.org/10.1007/BF00127126, 1991. Springer.

[MRA+20] Mohamed Massaoudi, Shady S. Refaat, Haitham Abu-Rub, Ines Chihi, and Fakhreddine S. Wesleti. A hybrid bayesian ridge regression-cwt-catboost model for pv power forecasting. https://doi.org/10.1109/KPEC47870.2020.9167596, 2020. 2020 IEEE Kansas Power and Energy Conference (KPEC).

[MRB+18] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P. Sheth. Machine learning for internet of things data analysis: a survey. https://doi.org/10.1016/j.dcan.2017.10.002, 2018.

[MRV+17] William M. Mongan, Ilhaan Rasheed, Khyati Ved, Shrenik Vora, Kapil Dandekar, and Genevieve Dion. On the use of radio frequency identification for continuous biomedical monitoring. https://ieeexplore.ieee.org/document/7946876, 2017. IEEE.

[Nad19] Mark Nadeski. Bringing machine learning to embedded systems. https://www.ti.com/lit/wp/sway020a/sway020a.pdf?ts=1681134429321&ref_url=https%253A%252F%252Fwww.google.com%252F, 2019. Texas Instruments.

[NAT+97] I. Nourbakhsh, D. Andre, C. Tomasi, and M. Genesereth. Mobile robot obstacle avoidance via depth from focus. https://doi.org/10.1016/S0921-8890(97)00051-1, 1997. Robotics and Autonomous Systems, vol. 22, pp. 151-158.

[Nil93] Nils J. Nilsson. *INTRODUCTION TO MACHINE LEARNING*. Stanford University, Stanford, CA 94305, 1993.

[PCC16] Chong Peng, Jie Cheng, and Qiang Chenga. Supervised learning model for high-dimensional and large-scale data. https://doi.org/10.1145/2972957, 2016. ACM.

[PMX+20] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, et al. Towards scalable dataframe systems. https://arxiv.org/pdf/2210.03519v1.pdf, 2020.

[PP16] Keyur K Patel and Sunil M Patel. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application, and future challenges. https://ijesc.org/upload/8e9af2eca2e1119b895544fd60c3b857.Internet%20of%20Things-IOT%20Definition,%20Characteristics,%20Architecture,%20Enabling%20Technologies,%20Application%20&%20Future%20Challenges.pdf, 2016.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, et al. Scikit-learn: machine learning in python. https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf, 2011. ACM.

[RAR21] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. Tinyol: tinyml with online-learning on microcontrollers. https://doi.org/10.1109/IJCNN52387.2021.9533927, 2021. IEEE.

[Ray20] Partha Pratim Ray. A review on tinyml: state-of-the-art and prospects. https://doi.org/10.1016/j.jksuci.2021.11.019, 2020. ACM.

[RDG+16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: unified, real-time object detection. https://doi.org/10.1109/CVPR.2016.91, 2016. IEEE.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: an incremental improvement. https://doi.org/10.48550/arXiv.1804.02767, 2018. University of Washington.

[RHG+15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: towards real-time object detection with region proposal networks. https://arxiv.org/pdf/1506.01497.pdf, 2015. Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, Cambridge, MA, USA, NIPS'15, pp. 91–99, MIT Press.

[RM05] Lior Rokach and Oded Maimon. Decision trees. https://doi.org/10.1007/0-387-25465-X_9, 2005. Springer.

[RRN02] Anthony Rowe, Charles Rosenberg, and Illah Nourbakhsh. A low cost embedded color vision system. http://dx.doi.org/10.1109/IRDS.2002.1041390, 2002. IEEE.

[RvdST+13] Uijlings J. R. R., van de Sande K. E. A., Gevers T., and Smeulders A. W. M. Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171, 2013.

[Sar21] Iqbal H. Sarker. Machine learning: algorithms, real-world applications and research directions. https://doi.org/10.1007/s42979-021-00592-x, 2021. Springer.

[SBW+97] R. Sargent, B. Bailey, C. Witty, and A. Wright. Dynamic object capture using fast vision tracking. https://doi.org/10.1609/aimag.v18i1.1275, 1997. AI Magazine, vol. 18, no. 1.

[SC08] I. Steinwart and A. Christmann. *Support vector machines*. Springer, 233 Spring Street, New York, NY 10013, USA, 2008.

[Sit21] Daniel Situnayake. How tensorflow helps edge impulse make ml accessible to embedded engineers. https://blog.tensorflow.org/2021/06/how-tensorflow-helps-edge-impulse-make-ml-accessible.html, 2021.

[SNdA+22] Gilberto Francisco Marthade Souza, Adherbal Caminada Netto, Arthur Henrique de Andrade Melani, Miguel Angelo de Carvalho Michalski, and Renan Favarãoda Silva. Engineering systems fault diagnosis methods. http://dx.doi.org/10.1016/B978-0-12-823521-8.00006-2, 2022. Elsevier.

[SRO15] Arnold Salas, Stephen J. Roberts, and Michael A. Osborne. A variational bayesian state-space approach to online passive-aggressive regression. https://arxiv.org/pdf/1509.02438.pdf, 2015. International Statisitical Review.

[TAH06] D. K. Tasoulis, N.M. Adams, and D. J. Hand. Unsupervised clustering in streaming data. http://dx.doi.org/10.1109/ICDMW.2006.165, 2006. IEEE.

[Tec15] AVAGO Technologies. Apds-9960, digital proximity, ambient light, rgb and gesture sensor. https://docs.broadcom.com/doc/AV02-4191EN, 2015.

[Tro19] Fabricio Troya. Detecting colors with the nano 33 ble sense board. https://docs.arduino.cc/tutorials/nano-33-ble-sense-rev2/rgb-sensor, 2019.

[UN00] I. Ulrich and I. Nourbakhsh. Appearance-based obstacle detection with monocular color vision. , 2000. Proceedings of AAAI Conference, pp. 866-871.

[VTP+20] Ms. A.Ramya Visalatchi, Ms. T.Navasri, Ms. P.Ranjanipriya, and Ms. R.Yogamathi. Intelligent vision with tensorflow using neural network algorithms. http://dx.doi.org/10.1109/ICCMC48092.2020.ICCMC-000175, 2020. IEEE.

[WN07] David Wipf and Srikantan Nagarajan. A new view of automatic relevance determination. https://proceedings.neurips.cc/paper/2007/file/9c01802ddb981e6bcfbec0f0516b8e35-Paper.pdf, 2007.

[WTS+20] Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng (Polo) Chau. Cnn explainer: learning convolutional neural networks with interactive visualization. https://doi.org/10.48550/arXiv.2004.15004, 2020. IEEE.

[WZZ+21] Lizhi Wanga, Zhaohui Zhanga, Xiaobo Zhanga, Xinxin Zhoua, Pengwei Wanga, and Yongjun Zheng. A deep-forest based approach for detecting fraudulent online transaction. https://doi.org/10.1016/bs.adcom.2020.10.001, 2021. Elsevier.

[YLJ18] Liang Yu, Binbin Li, and Bin Jiao. Research and implementation of cnn based on tensorflow. https://www.researchgate.net/journal/IOP-Conference-Series-Materials-Science-and-Engineering-1757-899X, 2018. IEEE.

[YYH18] Hao Yu, Mei Yan, and Xiwei Huang. Cmos image sensor. https://doi.org/10.1002/9781119218333.ch7, 2018. IEEE.

[Zac21] Zach. Mse vs. rmse: which metric should you use? https://www.statology.org/mse-vs-rmse/, 2021.

[ZKK+19] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A review of machine learning and iot in smart transportation. https://www.mdpi.com/1999-5903/11/4/94, 2019. MDPI.

[ZW18] Zhenchong Zhao and Xiaodan Wang. Multi-segments naïve bayes classifier in likelihood space. https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/iet-cvi.2017.0546, 2018.